A Possible Time-Series Object

Don Sawyer, Andrew Davis, John Garrett, Carrie Gonzalez, David Han, Joe King, Mike Martin,
Tom McGlynn, and Formats Evolution Process Committee

Version 14, 00-12-06
http://ssdoo.gsfc.nasa.gov/nost/fepc/time-series-object.html

Table of Contents

1. Objectives and Approach

At least two aspects of current "standard formats" and their associated software impede science analyses. On the one hand, formats may not have all the needed or desired functionalities. On the other hand, the multiplicity of formats renders multi-source data analyses (where various sources chose to cast data in various formats) very difficult.

A Formats Evolution Process has been underway for many months now to try to address and possibly get beyond these issues. See http://ssdoo.gsfc.nasa.gov/nost/fepc/. One activity of this process has been to define a "time series object" to better understand some of the underlying issues.

The purpose of this paper is to identify a possible definition for a general time-series data object, and to explore some of the implications for its implementation in several different formats that have been used for multiple science data sets. This effort provides some insights into the underlying formats themselves, and the degrees to which they need to be extended to support a general time-series data object. Further, if the community decides that defining a general time-series data object is a useful thrust, there could be a subsequent time-series data object standardization effort that should result in the following benefits:

1. Data product developers will know what metadata is needed to support the standard time-series view of their data, and including it will enable new data products to be more useful to researchers using existing application software.

2. There can be a set of standard interfaces supported by formatting system software (e.g., possible extensions to, or built on top of, HDF, CDF, IDFS), which then allows applications to work in a standard way with time series data from a variety of instruments and across disciplines.

3. The understanding of the time aspects of data made available as a time-series object will already be known to data users/researchers and will therefor reduce their learning time in handling new data products.

The Formats Evolution Process Committee (FEPC) (see http://ssdoo.gsfc.nasa.gov/nost/fepc/) has developed this draft 'time-series' object as a part of its overall objective of improving the ability of the science community to exchange, use, and preserve information in the context of multiple formats. The FEPC does not expect this work to be a standard, but to point the way if this proves productive.

1.1 How to Read the Paper

A single time-series object (TSO) can not do all that anyone might want. Each potential user may have some favorite requirements to be fulfilled. Necessarily a particular conceptual view of the important aspects of a TSO are needed. This is presented in section 2.1 and should be of interest to all readers. Then, a list of questions that a TSO might be expected to be able to answer is postulated in section 2.2. The TSO might answer these question by virtue of including the necessary information within. It might also answer these questions from an application API point of view, in the form of what inputs should one be able to give it and what type of results should one expect to get back. Our approach focuses on a minimum set of time-related services, but allows additional services to be added as needed. This helps identify the minimum metadata that is needed for a TSO and should be of interest to all readers including science end users and software developers. A full standardization effort would flesh this out more thoroughly and it may or may not include a standard API. Appendix A provides potential responses to the questions of section 2.1 from the perspectives of the FITS and HDF4 implementations.

Section 3 provides summarized examples of how a TSO might be implemented using some well known formats. Full details are given in Appendix B. Those who are familiar with one or more of these formats may be especially interested to see the implications of the approaches taken.

Section 4 gives insights from the implementations and should be of broad interest.

Section 5 provides an introductory views of how various APIs for a TSO may appear. First it provides a view of a set of low level APIs that allow programmers to manipulate the TSO and the various fields that comprise it. Then, a high level API is provided that supports directly finding the answers to the key TSO questions identified in section 2. Section 5 is particularly interesting to those who implement access software, and those who would be interested in using such TSO specific software. The detailed API specifications are given in Appendix C.

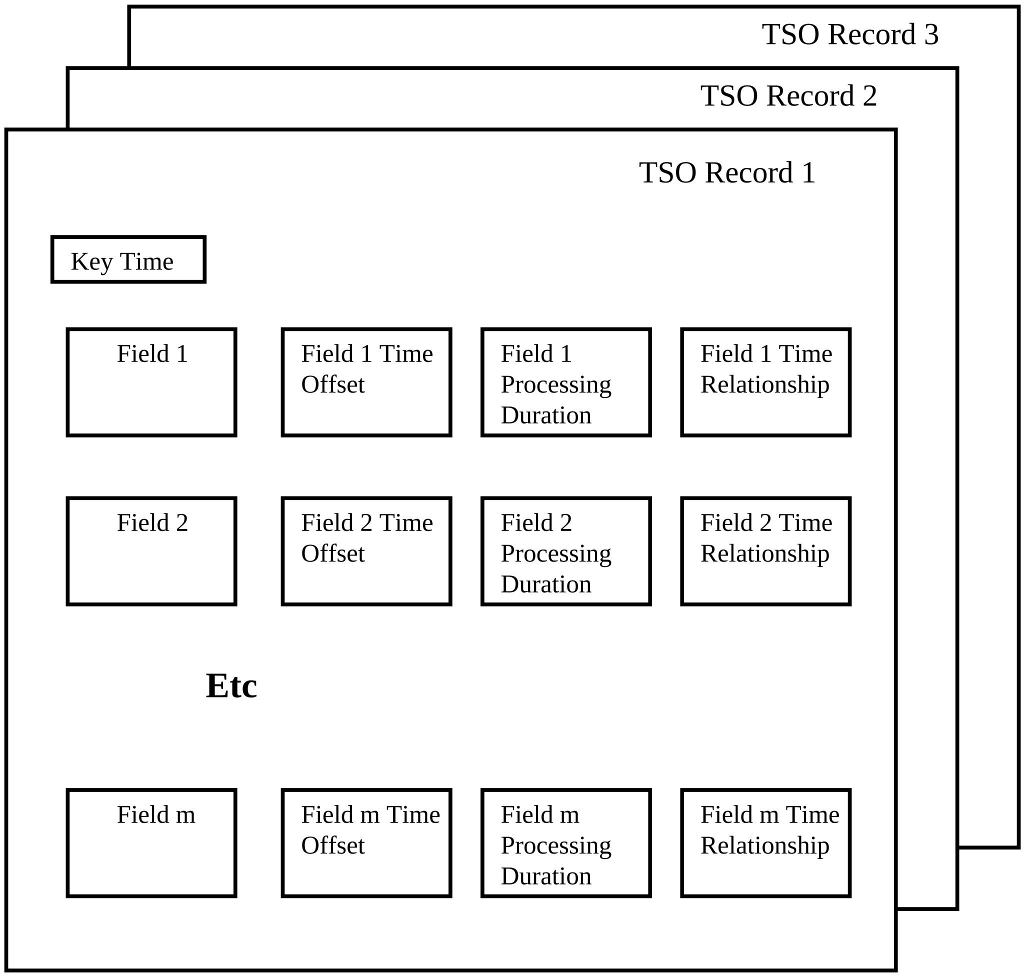
1.2 Responses to the Paper

Comments on all aspects of this paper are of great interest to the FEPC. They can be sent to fepc@nssdc.gsfc.nasa.gov, or to any of the individual FEPC members listed on the FEP Web site.

2. Time-Series Object

2.1 Conceptual View

It is important to note that the material to follow provides a conceptual view and is not an implementation. How the necessary information is actually organized and stored is not addressed in this section.



TSO Record 3
TSO Record 2
TSO Record 1
Key Time
Field 1
Field 1 Time Offset
Field 1 Processing Duration
Field 1 Time Relationship
Field 2
Field 2 Time Offset
Field 2 Processing Duration
Field 2 Time Relationship
Etc
Field m
Field m Time Offset
Field m Processing Duration
Field m Time Relationship

Figure 1: Time-Series Object Conceptual View

1. A Time-Series Object appears (conceptually) as a sequence of records, as shown in Figure 1. (Note: It may not be implemented as a simple sequence of records, however.)

2. Each record contains a 'key time' value (including date), which is monotonically increasing or decreasing from record to record throughout the sequence.

3. Each record also contains other data, of arbitrary complexity, which may vary from record to record. This data can be viewed as a sequence of fields, but may also be viewed as groups of fields which map to various types of objects such as images, vectors, matrices, etc. It is not necessary to call out all detailed structure, so that a whole image may be viewed as one field if so desired.

4. Each record also contains one or more 'processing durations'. Each such processing duration gives the time span over which one or more of the other data fields has been determined. For example, a data field may be a 1 minute average of other values, in which case the 'processing duration' would be 1 minute. It may, of course, be zero as well. The 'processing duration' is known for all data fields, identified as such, in the time-series object.

5. Each record also contains one or more 'time offset' values. Each data field has a time offset value which relates, through the use of addition, its time of observation/determination to the record's key time value. (See Figure 2 for an example.)

6. The relationship of the key time, plus time offset, to each fields processing duration is known because either it is given explicitly or it is a part of the standard. Possible relationships include:
- For each subset of the data fields, the key time, after being augmented with the relevant time offset value, corresponds to the beginning, the middle, or the end of the associated processing duration value. Figure 2 is an example where the offset is to the middle of the processing duration for data fields 'x' and 'y', but to the beginning of the processing duration for data field 'z'.
- Other ways to define the relationships are possible, but are not explored here.

7. The meaning of each field, of key time, of processing durations, of time offsets, and the units associated with these entities, is known and described. This is a general requirement intended to ensure that the information that comprises the TSO is at least minimally understandable.

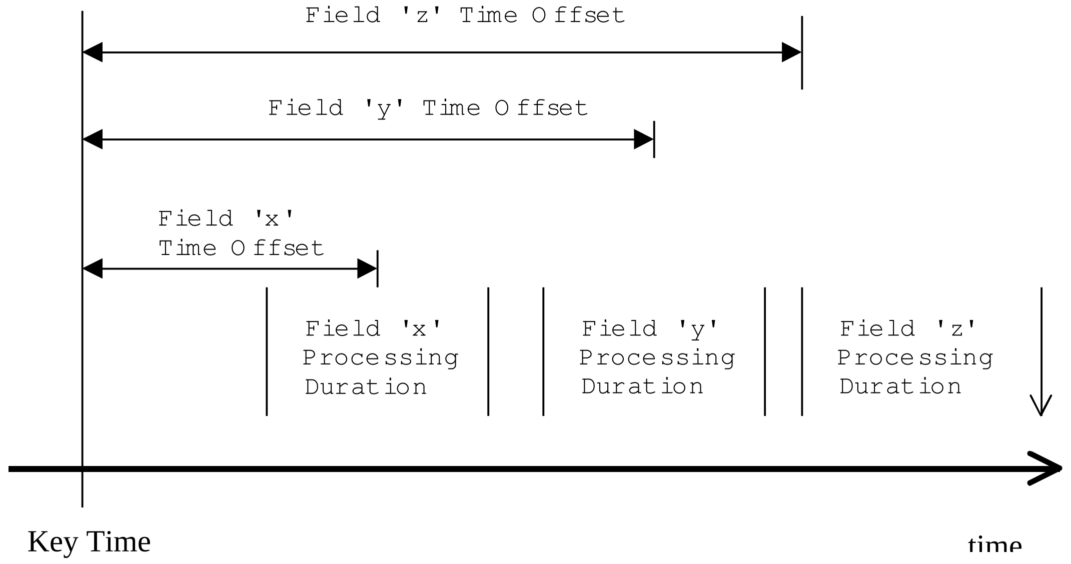

Field 'z' Time Offset
Field 'y' Time Offset
Field 'x' Time Offset
Field 'x' Processing Duration
Field 'y' Processing Duration
Field 'z' Processing Duration
Key Time
time

Figure 2: Time Relationships Example

2.2 Questions to be Supported

These questions can be viewed as motivating the necessary data or metadata to be available as a part of the time-series object, and/or as suggesting the types of values that an API might require.

2.2.1 Proposed Required Questions for Any Data Object

1. What are the fields available (when looking across all the records)? (Give names)

2. What is the definition of field 'x'.

3. What is the representation type for field 'x' (e.g., integer, real, image array, character string)

4. What are the units for field 'x'?

2.2.2 Proposed Required Questions for Time-series Object

1. What is the meaning of the key time as used in this object?

2. What is the start date/time and what is the end date/time for this object?

3. What is the processing duration for data field 'x'?

4. What is the offset time for data field 'x'?

5. How does the offset time relate to the processing duration for data field 'x'? (e.g., 'mid point')

6. Get the records for key times between time a and time b.

7. Get the values for fields 'x' and 'y' for the key tims between time a and time b.

2.2.3 Proposed Optional Questions for Time-series Object

1. Get the values for fields 'x' and 'y' when their individual times are between time a and time b.

2. How many records exist between time a and time b?

3. What is the maximum record size in bytes?

4. How many unique data field types are present in the implemented object?

3. Example Implementations Summarized

In order to clarify both conceptual and practical issues, this section provides some summarized examples of how a TSO might be implemented using a few data formats. For illustrative purposes, a model data set is defined in section 3.1 and then mapped into the TSO conceptual view in section 3.2. Sections 3.3 through 3.8 provide summaries of example implementations of the TSO model data in various formats. The full examples are given in Appendix B. These implementations reflect only some of the ways these formats could be used to hold the TSO model data.

3.1 Model Data Set

The data set consists of the following:

- One year of data.
- One day of data per data file.
- One minute of data in each data record.
- Gaps in the data, where minutes with no data have no records.

The data in the record are:

Word	Name	Further definition
1.	time tag	say, YYYYDDDtHHMM
2.	s/c location	X,Y,Z vector at time of time tag
3.	count rate 1	hypothetically instantaneously determined count rate at 1st of 12 energy steps, determined at the record's time tag
4.-14.	count rates 2-12	rates determined at 2rd-12th energy steps, each offset by 5 sec from measurement of prior step
15.	density	determined from rates 1-12 by taking moments of distribution function.

Word 1 is ASCII, word 2 is (3x)R*4, words 3-14 are I*4, and word 15 is R*4.

Relative to offsets from the record's time tag, words 3-14 have offsets of 0, 5, 10, ... 55 sec. For word 15 the record's time tag designates the start time (not mid or end or other time) for word 15's "processing interval."

3.2 Partial Mapping to the Time Series Object View

The model data set of section 3.1 is mapped to a TSO view to provide a consistent baseline for the implementations that follow. Where the processing duration is zero, the relationship of the duration to the associated time is arbitrarily set to 'Begin of Duration' . No assumption is made as to whether there will be one or multiple TSO objects resulting from the application of the TSO view to the model data. This is an implementation decision.

- TSO record = data set record
- TSO = sequence of TSO records, covering the year (Note: Gaps in the data result in gaps in the TSO record sequence. Actual implementations may handle gaps differently.)
- Time Order = ordered with increasing values of the time tag
- Key Time = time tag

TSO Field Matrix:

Field Name	Offset Int.	Processing Duration	Relationship of time To Proc. Duration
S/C Location			
X value	0 sec	0 sec	"Begin of Duration"
Y value	0	0	"Begin of Duration"
Z value	0	0	"Begin of Duration"
Count Rate 1	0 sec	0 sec	"Begin of Duration"
Count Rate 2	5 sec	0 sec	"Begin of Duration"
Count Rate 3	10 sec	0 sec	"Begin of Duration"
.			
.			
.			
Count Rate 12	55 sec	0 sec	"Begin of Duration"
Density	0 sec	55 sec	"Begin of Duration"

3.3 CDF Implementation

The model data set is implemented using CDF's (Common Data Format) Variables and Attributes. Variable is an entity that represents/contains data, and attribute is the mechanism for storing metadata. There are two types of attributes in CDF: global and variable. Global attributes are used to describe a CDF file/data set (e.g. data set creator, file history, etc.) and to hold values that are common/global to the data set. Similarly, variable attributes are used to describe or provide additional information about variables. One or more variable attributes can be attached to a variable. For example, a variable can have MIN and MAX attributes to represent the minimum and maximum values allowed for this variable, and it can have a text description of the variable.

3.3.1 CDF/TSO Summary

TSO Objects: The TSO object is implemented as a single CDF file containing the year's data, or it could be broken into multiple TSO objects covering fractions of the year. It simply depends on how much data is loaded into a given CDF file.

TSO Object Identification: The TSO object is identified as such by a global attribute "ObjectTypes".

TSO Records: A TSO record is comprised of a CDF record containing the defined variables with their attributes, together with the CDF global attributes needed to complete the TSO record information. Note that each TSO record logically incorporates the same global attributes. The CDF record has one variable for each of spacecraft position, count rate, and density. The meaning and units for these variables are given by the associated variable attributes.

TSO Time Order: TSO time order is given by a CDF global attribute.

TSO Key Time: Key Time is given by a CDF character variable KeyTime. A global attribute is used to state that the time order is 'increasing'. KeyTime meaning is given by an associated attribute.

TSO Offsets: TSO offsets are implemented as CDF variable attributes for the position vectors and for the density. For the count rates, an attribute is used to specify the offset for the first of the 12 rates, and offsets for the other rates must be calculated by using the 'time increment' attribute associated with the count rate variable. The units for the offsets, and their meaning, are given by an associated global attribute. The association is given by pre-pending a 'g' to form the global attribute name. This is a local convention.

TSO Processing Durations: TSO processing durations are implemented as CDF variable attributes for position, count rate, and density. The units for the durations, and their meaning, are given by an associated global attribute. The association is given by pre-pending a 'g' to form the global attribute name. This is a local convention.

TSO Field/Time Relationships: TSO field/time relationships are implemented by CDF variable attributes for position, count rate, and density. The allowed values for the relationships, which are reasonably meaningful, are given by an associated global attribute. The association is given by pre-pending a 'g' to form the global attribute name. This is a local convention.

Other: Missing data are given some type of fill value, not defined here.

3.4 HDF4 Implementation

We use two HDF building blocks to implement the TSO in HDF - Vgroups and Vdatas. Vgroups are generic grouping elements allowing a user to associate related objects within an HDF file. As Vgroups can contain other Vgroups, it is possible to build a hierarchical file. Vdatas are generic list objects. Data is organized into "fields" within each Vdata. Each field is identified by a unique "field name". The type of each field may be any of the basic types that HDF supports. Fields of different types may exist within the same Vdata.

For this example, Vgroups are not very important. If there were data from several different instruments within the data file, one could define a Vgroup for each instrument, for example. In any case, within a Vgroup we can define a Vdata, and within the Vdata we can define the fields into which the data is to be organized. So, there will be a correspondence between the Vdata fields and the Words in the model data set.

We use HDF File Annotations and HDF Object Annotations and HDF Attributes to record metadata within the HDF files. Any generic HDF data browser tool should be able to display all annotations and attributes contained within any HDF file.

HDF annotations are basically containers into which one can dump short text descriptions. File annotations are used to record global information about the dataset. Object annotations can be attached to Vgroups or Vdatas or other objects within a HDF file. We would attach an object annotation to the TSO Vdata that would describe the Vdata fields.

HDF attributes each have a name, a data type, a number of attribute values, and the attribute values themselves. Any number of attributes can be assigned to either a Vdata or any field within a Vdata, as long as they are named uniquely.

We would use HDF attributes to define units for the Vdata fields, offsets for the count-rates, MAX/MIN limits, etc.

3.4.1 HDF4 Summary

TSO Objects: The TSO object is implemented as a single HDF4 file containing the year's data, or it could be broken into multiple TSO objects covering smaller time periods.

TSO Object Identification: HDF files can contain more than one kind of Vgroup or Vdata. The TSO object is identified as such by the name given to the TSO Vdata when when the HDF file is written, and by the Vdata annotation.

TSO Records: A TSO record is comprised of a HDF Vdata instance containing the defined fields. The field attributes, and the file and object annotations round out the information needed to complete the TSO record. The meaning and units for the Vdata fields are given by the associated attributes, and the Vdata annotation.

TSO Time Order: TSO time order could be specified in the Vdata annotation. However, that would not actually stop someone from writing out records in whatever time order she wishes. There is nothing special about the time fields in the Vdata.

TSO Key Time: Key Time is given by the a Vdata fields year, day, hr, min. Additionally, the fields fp_year, fp_doy, and Epoch are available.

TSO Offsets: TSO offsets are implemented as HDF attributes for the position vector components, the count rates, and the density.

TSO Processing Durations and Field/Time Relationships: TSO processing durations and Field/Time Relationships are implemented as HDF attributes.

Other: Missing data can be handled by inserting records containing fill data for all but the Key Time variables, or not. The particular approach taken would be documented in the annotations.

3.5 IDFS Implementation

There are various ways in which the model data set can be stored under IDFS, which is probably a statement that can be made by other formats as well. Under IDFS, you store the RAW data and the procedures to convert that raw data into geophysical quantities. That is not to say that computed data cannot be stored under IDFS; it is simply that the strength of IDFS is in the ability to modify the procedure for data conversion into physical units without the need to re-process all data sets since the raw data is what is stored.

With IDFS, data is grouped into virtual (or logical) instruments. A virtual instrument is a group of sensors that are linked together by commonality; therefore, it makes sense to treat the sensors together. A virtual instrument can be classified as either a scalar instrument or a vector instrument. A scalar instrument returns singular data quantities that are dependent only upon time and position. An IDFS vector instrument returns one-dimensional data quantities that have a functional dependence upon a single variable called the scan variable. The length of this 1-D vector is virtual instrument dependent.

There appears to be three different types of data contained in the model data set: (1) the s/c location (which are three scalar values), (2) the count rates consisting of 12 elements, and (3) density (a scalar processed value). It is assumed that the spacecraft location and count rates are measurements with discrete values. These data values should be stored in their raw form and the IDFS should contain the necessary descriptions for converting the stored data numbers into geophysical numbers. If the assumption is not correct and these are all computed quantities, they can be stored as is within IDFS as data.

One possible scheme would be to subdivide the data into 3 virtual instruments, which are referred to as (1) POSITION, (2) COUNT_RATE, and (3) DENSITY. In the IDFS paradigm, each data set is written into two types of files: the header file (H) and the data file (D). File names link the proper header and data files together. There is also a VIDF file and a PIDF file per virtual instrument.

The Virtual Instrument Description File (VIDF) is a complete description of the virtual instrument. The VIDF file is meant to be easily updated and to contain all of the data that may be periodically updated due to either refinement in the instrument calibration or due to the degradation within the instrument. There must be at least one VIDF file defined for each IDFS virtual instrument. If data within the VIDF changes with time, for example calibration coefficients, additional VIDF files can be defined. The VIDF file provides a general description of the measurements being stored in IDFS format, and contains the data reconstruction parameters that are needed in order to transform the raw data into physical units. In addition, the VIDF file provides information by which data from the data file can be extracted, such as the size of data records.

The Plot Interface Definition File (PIDF) is an optional file under the IDFS format. The word optional in this context means that it is not necessary in the use of any of the IDFS data access software; that is, data in IDFS format can be located through the database, accessed and converted to units without the PIDF file. If, however, you plan to use any IDFS based data display or analysis software that has been developed by Southwest Research Institute (SwRI), the PIDF file is required.

The PIDF is best described as an interface file between the IDFS VIDF definitions and a general user interface to a display or analysis program, providing a large number of display attributes. The PIDF file is an ASCII file that describes how to display the data in a meaningful way. The PIDF file defines the units that are available for each sensor including the correct VIDF tables to apply, how they are applied and the scaling limits (min/max) for each set of units.

The header files contain data which, for the most part, is slowly varying in time and need not be repeated every data record. Each IDFS data record points to a header record that describes the state of the instrument at that particular point in time.

The vast majority of all of the telemetered data is stored within the data file, which contains the most rapidly varying data. The data records contain the base time tag for the data and raw, unprocessed binary data. Unlike the header record, the data record does not vary in size. The size is specified in the VIDF file.

3.5.1 IDFS Summary

TSO Objects: The model data would be split into three separate TSO objects corresponding to spacecraft position, count rates and density. Each TSO object would be comprised of four file types (data, header, VIDF and PIDF). There would be one data and one header file, each containing one-year's worth of data. Other time intervals could be utilized, if the implementer saw the need for smaller files. There would be one VIDF and one PIDF file defined for the whole time duration of the model data set.

TSO Object Identification: Currently, there is no field or attribute that would identify a TSO object. However, the VIDF file is "extensible" so additional identifiers (attributes) could be added while preserving backwards compatibility for data sets already in IDFS that would not have the newly defined identifiers.

TSO Records: A TSO record corresponds to the set consisting of a data record, the corresponding header record, and information that is extracted from the VIDF file.

TSO Time Order: Within the IDFS paradigm, time does not have to be monotonically increasing or decreasing; therefore, there is no attribute of this type in IDFS. Data that is jumping around in time can be stored under IDFS. That is not to say that this scenario wouldn't be a nuisance for the applications that are processing the data.

TSO Key Time: Key Time is split between the header record (YYY, DDD) and the data record (milliseconds of the day).

TSO Offsets: TSO offsets are placed in the VIDF file and are utilized when the time tag is computed. An exception is noted for the 12 count rate values, which were combined into a single IDFS vector sensor. For the count rate values, the time offset values of 5 seconds for each element of the vector in the VIDF file is split between the built-in attributes data_accum and data_lat, with very little going to data_accum as the rates are taken to be near instantaneous observations.

TSO Processing Durations: TSO processing durations are handled by the built-in attribute time_units, data_accum, data_lat, swp_reset and sen_reset. The set of values describes the current accumulation times and is used to correctly time tag the given data values found within the data records.

TSO Field/Time Relationships: TSO field/time relationships are implicitly defined in IDFS. Within IDFS, the relationship is always defined as corresponding to the start (beginning) of the processing duration value. Therefore, there is no explicit attribute or identifier to express the relationship. Since the VIDF file is "extensible", an additional identifier (attribute) could be added which would simply always be set to "start of duration".

Other: Gaps in the data are handled by missing records or by fill values, as defined in the VIDF file.

3.6 FITS Implementation

The FITS data format is defined solely in terms of the data structure as stored on disk or tape media. No standard FITS API exists although the format is nearly universally supported in astronomy analysis software.

A FITS file consists of one or more Header-Data Units (HDUS). Descriptive information about the data is placed in the header which completely defines the structure of the data that follows. The header may optionally contain information on the semantics of the data, i.e., the unit, definitions of the axes and so forth.

FITS addresses the relationships between the numbers in the FITS files and the physical units in several ways. For some keywords values, e.g., coordinates, the units are defined in the FITS standard. For other keywords the units are often given using comments. The relationship between data elements and phyical units is more completely detailed using the FITS WCS conventions which have now been submitted for formal ratification by appropriate national committees. For scalar elements of tables, there are explicit scale, offset and unit keywords. For non-scalars -- either in tables or images -- FITS allows the user to associate each dimension (viewed as pixels) of the object with a physical axis where appropriate keywords describe a linear transformation from the pixel offsets to physical units.

For the reference data set, an implementation as a FITS table, or a set of tables is most natural. We have implemented the reference data set using the same file structure as in the reference model, with each day kept as a separate file. It would be perfectly feasible to organize all of the data in a single file, or to have multiple HDU's with different days within a single FITS file.

To realize a FITS file with these data we would need to create a file with two HDU's. The first is a dummy HDU where the header can contain descriptive information about the file, but will contain no actual data. Table data is not permitted in the first HDU. The header for an HDU comprises a series of 80 byte records terminated by an END record.

Following the primary HDU (with some padding required by rules originally designed to enable FITS to work well with tape media) the header for the second HDU is written. This consists of a series of required keywords which indicate the type and size of the table, and then definitions of each of the columns. Only the actual data format of the column is required but much other information about the column can be given.

After a small amount of padding the data would be written in big-endian, IEEE standard format.

3.6.1 FITS Summary

TSO Objects: The FITS standard describes only a data format. There is no standardized API to the system and thus there are no FITS ‘objects’ as such. However FITS does allow for encapsulation and organization of data in a series of FITS records. FITS does provide for several keywords (notable the EXTNAME) keyword which can be used to define the type of data being described. The HDUCLASS hierarchy convention used at the HEASARC is easily supported by FITS. This provides for a loose inheritance hierarchy, but this is not a general standard. In this implementation the model data are inserted into files, one per day. However they could have all been put into one file either as separate HDU's within the file or as one large HDU.

TSO Object Identification: A HDU keyword could be defined for the first (dummy) HDU to carry the information that this implementation is a TSO object.

TSO Records: FITS data is organized as a series of header-data units where each of these has a standard structure - either a table or a multi-dimensional array. Each of these can be characterized by a set of keywords in the header. Within each HDU a table may have any number of rows. The TSO record corresponds to a HDU in this implementation.

TSO Time Order: For FITS tables time order can be specified in the headers using either an start time and offset for each record - in which case the data must be monotonic (though possibly in either direction) - or times can be specified explicitly for each record in which case data can be in any order. The actual TSO order, whether increasing or decreasing, could be specified by a new keyword in the first HDU.

TSO Key Time: Typically FITS tables include a key time column whose format is described in ancillary metadata. The TSO key time is given by the first field of a table row and is in seconds since 1980 (HEASARC convention).

TSO Offsets: Offsets for individual data within a record can be set using the nCDLTnn keywords.

TSO Processing Durations: FITS does not explicitly address series of non-contiguous pixels within a record. However if a record contains a scalar value a combination of the nCDLT and nCRPX keywords can simulate a single interval which begins and ends at an arbitrary offset from the key time for the record.

TSO Field/Time Relationships: The nCDLT, nCRPX, and nCRVL describe the full relationship between the field and the key time. However as mentioned above if the field is a vector in time, FITS does not provide for gaps between the elements of the vector.

Other: There are no general conventions for missing data, although FITS tables have a convention for NULL values and binary tables explicitly support the use of IEEE NaN's (Not-a-Number) for that purpose.

3.7 PDS Labels Implementation

The PDS label architecture is oriented to providing an archival description of a data file, rather than to supporting display or manipulation of the contents. The PDS SERIES object which is used in this example is a slight specialization of the PDS TABLE object, which is just a flat file of ASCII or binary values (depending on the value of the INTERCHANGE_FORMAT keyword).

TSO implementation.

Offsets. These are normally specified in descriptive text. The sampling parameter keywords can also be used to indicate a repeating group of regular observations as illustrated in the example.

File organization. The organization of data into physical files in the PDS architecture is done using a set of PDS guidelines based on size and time. In this case each physical file is specified to be one day’s worth of data records or 1440 records. There would then be conventions for directory names and file names that would provide convenient access to a file containing a certain period of time.

Units. Units can be specified for a column object using the UNIT keyword. When units need to be associated with a sampling parameter, the SAMPLING_PARAMETER_UNIT keyword is used.

Time associations. In general, time associations are specified within the description of a field. There are duration keywords which can be used to indicate some time associations, like the frame_duration which indicates an instantaneous measurement in the example.

Drop outs. There are no conventions for handling drop outs.

3.7.1 PDS Labels Summary

TSO Objects: The model data would be represented as a sequence of time series objects, with each implemented as a file pair. One file carries the metadata attributes for the other file. The other file carries one day's data where each record represents one minute of data. Directory and file name conventions would tie the individual TSO objects into a single logical TSO for the year of data.

TSO Object Identification: The object is a TIME_SERIES which is a variant of TABLE. The TIME_SERIES_TYPE keyword indicates that this is a FEPC_TSO.

TSO Records: A TSO record corresponds to a single row in the TIME_SERIES table. It stores 12 measurements and represents one minute of data.

TSO Time Order: The keyword TIME_ORDER_TYPE = ASCENDING, DESCENDING, N/A has been added to indicate time order.

TSO Key Time: The keyword KEY_COLUMN = START_TIME is added to indicate that it is the key field.

TSO Offsets: The offset of 5 seconds between count rate measurements is specified by the sampling parameter keywords in the COUNT_RATE column.

TSO Processing Durations: PDS uses the keyword FRAME_DURATION. Durations are specified in the field description for COUNT_RATE and DENSITY.

TSO Field/Time Relationships: TSO field/time relationships are specified with the sampling parameter keyword embedded in the column definition. In the example the count rate has a sampling parameter and the density has a sampling parameter. There is no way to specify two sampling parameters (say time and energy) for a single column.

Other: Gaps in the data result in missing records.

4. Implementation Insights

It is clear that all the formats investigated, and probably most any others, are able to be extended to support the concept of this general time-series object. What is of particular interest is the extent and approaches by which the various formats are extended to meet this objective. While this can not be fully investigated without actual prototyping, there are some insights which emerge from the paper study. Further insights will no doubt arise from community review.

For some formats, such as CDF, there is no standard way to associate additional metadata,, such as units or valid values, with attributes already associated with the primary data values. This may be seen as an 'attributes of attributes' issue. In the case of the CDF implementation, this is overcome by assigning additional global attributes and in their multi-part values also indicating with which other attributes they are to be associated. In effect this extends the basic CDF data model but of course it will not be recognized as such by the current CDF software.

From the HDF4 implementation, it is noted that in order to easily provide units in association with the attributes, the attributes are shown as text. To make the attribute values more accessible to manipulation, they could have been implemented as integers. In this case some new global attributes could be used to relate the vgroup attributes with their units, much as was done with the CDF implementation example. Current HDF4 software would, or course, not recognize these new relationships.

From the PDS implementation, it was noted that software has not been implemented for all the PDS objects. Therefore some of the needed information, such as relating sampling_parameter to count rates, is not given explicitly or it may be incorporated into text description where it is not readily accessible to software manipulation. This particular relationship could be implied by including the sampling_parameter inside the count rate object.

IDFS is unique among the formats implemented in at least two respects. First, there are at least three files for every TSO object and it takes multiple (three) TSO objects to hold all the TSO model data for any given time interval. This increased complexity, however, gives it considerable flexibility. Second, IDFS has built in capabilities for handling all the major components of the TSO model. About all that needs to be added are a couple of attributes to say that the TSO is present, and to indicate whether the data were increasing or decreasing in time. IDFS has associated software that can combine the 3 TSO objects into a single 'derived' TSO and this would likely form the basis for supporting a TSO API.

The FITS implementation takes advantage of the additional standardization prescribed by the HEASARC project. As such, apart from a few additional keywords needed, it pretty well captures the TSO. Since FITS does not have standardized APIs, TSO software access would be built on existing implementations.

All of the formats, except PDS Labels, used as test implementations result in some re-formatting of the original model data. It would also be interesting to explore the use of a more generic data description language, such as EAST, in supporting a TSO.

5. Time-series API Introduction

[Note: For convenience of our readers, we have repeated Section 5 at the beginning of Appendix C so that all the API information is available in one place.]

This section describes possible Application Programming Interfaces (APIs) that can be used to accommodate the Time Series Object (TSO) requirements defined in section 2. It has been estimated that for each of the formats discussed in section 3, the existing software support could likely be the basis for the types of services described below.

An Object Oriented (OO) approach is employed in designing the APIs with the following requirements and assumptions:

- Support simultaneous access to multiple TSOs.
- Data is organized and stored as described in Figure 1 (Time-Series Object Conceptual View) of section 2 , except the actual data values are viewed as trailing the time relationships for each record field.
- The structure of each TSO record is consistent

Sections 5.1 and 5.2 provide an introduction to the detailed API definitions specified in Appendix C. A fully consistent set of API definitions has not been attempted, but there is enough to give a good idea of the approach.

Changes to be completed in future versions would include:

♦ In Section 2, we indicate that record sizes can vary. However some of the current API calls assume the record size is constant, i.e. each record (logically) contains an instance for every data field. If record size varied, then the data fields available in each record would vary. We may choose to provide an API either for vary size records or fixed sized records or perhaps even a API version for each case. The API version will affect what parameters are needed on inputs for many of the TSOField calls. We will likely need to update parameters for some of the calls whichever version of the API is provided.

♦ If data fields can vary within records or if the order of the data fields within a record are unknown, then additional data field identification and/or data type information will be needed in a number of the TSO field calls.

5.1 Application Programming Interfaces (APIs)

Each API returns a status code of Integer*4 to indicate whether the API is successfully completed. If the API is not successfully completed, it should return a negative number. Otherwise it should return the status code value of greater than or equal to 0 to indicate the successful completion of the API. Status codes and their explanation texts are not documented in this document since they can vary from implementation to implementation. The following describes the meaning for each of the data types used in the APIs.

Data Type	Description
String	A character string that has one or more characters. The size of a string is indicated by adding a colon after the word'String' followed by the string size. For example, String:20 represents a string that is 20 characters long.
Integer*4	4-byte signed integer
Real*4	4-byte floating point number
Void	Data type used to represent any of the data types described above. This data type is used to send or receive data to/from an API and is equivalent to 'void' in C, 'equivalence' in Fortran, and 'Object" in Java.

There are two categories of APIs: low level APIs and high level APIs. Low level APIs provide a basic set of functions that allow users to create and manipulate TSOs, TSO records, and TSO record fields. High level APIs allow users to perform more sophisticated functions such as "get the records whose key time is between time A and time B", "get the time range (begin and end time) for the given TSO", "get the values for fields 'x' and 'y' for the key time between time A and time B", and etc. High level APIs make use of low level APIs as building blocks, often in combination with some programming.

Naming Conventions

- Calls are mixed case with the first letter of each word capitalized all other letters are lower case. (E.g., GetKeyTime, GetTSORec) (Note TSO is an abbreviation and is always capitalized.)
- Calling Parameters are mixed case, the first letter of each word except the first is capitalized, all other letters are lower case. (E.g., name, recNum, TSOId) (Note TSO is an abbreviation and is always capitalized.)

- The following abbreviations are used consistently throughout in names of Calls and Calling Parameters. These abbreviations are capitalized according to naming convention rules, except that TSO is always in all upper case.
  - Id - Identifier
  - Num - Number
  - Processing - Proc
  - Rec - Record
  - TSO - Time Series Object
- Calls named as CreateXxx will have a matching DeleteXxx call.
- Calls named as SetXxx will have a matching GetXxx call.

Low Level APIs

TSO Handling APIs

- CreateTSO (String:30 name, Integer*4 TSOId)
- DeleteTSO (Integer*4 TSOId)
- AskTSOId (String:30 name, Integer*4 TSOId)
- AskTSOName (Integer*4 TSOId, String:30 name)

TSO Record Handling APIs

- CreateTSORec (Integer*4 TSOId, Integer*4 recNum)
- DeleteTSORec (Integer*4 TSOId, Integer*4 recNum)
- AskTSORecSize (Integer*4 TSOId, Integer*4 recNum, Integer*4 recSize)
- SetKeyTime (Integer*4 TSOId, Integer*4 recNum, String:22 keyTime)
- GetKeyTime (Integer*4 TSOId, Integer*4 recNum, String:22 keyTime)
- SetTSORec (Integer*4 TSOId, Integer*4 recNum, String:variable TSORec)
- GetTSORec (Integer*4 TSOId, Integer*4 recNum, String:variable TSORec)

TSO Record Field Handling APIs

- CreateTSOField (Integer*4 TSOId, Integer*4 recNum, String:30 name, Integer*4 dataType, Integer*4 numDims, Integer*4 dimSizes[], Integer*4 fieldId)
- DeleteTSOField (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId)
- SetTSOField (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldID, Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)

- GetTSOField (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)
- SetTSOFieldData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldID, Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)
- GetTSOFieldData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)
- SetTimeOffset (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 timeOffset)
- GetTimeOffset (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 timeOffset)
- SetProcessingDuration Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 procDuration)
- GetProcessingDuration (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 procDuration)
- SetTimeRelationship (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, String:18 timeRelationship)
- GetTimeRelationship (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, String:18 timeRelationship)
- SetData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Void data)
- GetData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Void data)
- AskTSOFieldId (String:30 name, Integer*4 fieldId)
- AskTSOFieldName (Integer*4 fieldId, String:30 name)
- AskTSOFieldType (Integer*4 fieldId, Integer*4 dataType, Integer*4 numDims, Integer*4 dimSizes[])

High Level APIs

TSO Record Handling APIs

- ExtractTSORecs (Integer*4 TSOId, String:22 startTime, String:22 endTime,String:variable TSORecs[])
- AskTotalTSORecs (Integer*4 TSOId, Integer*4 numRecs)
- AskNumTSORecs (Integer*4 TSOId, String:22 startTime, String:22 endTime, Integer*4 numRecs)

TSO Record Field Handling APIs

- AskTSOFieldNames (Integer*4 TSOId, String:30 names[])
- AskTSOFieldIds (Integer*4 TSOId, Integer*4 fieldIds[])
- AskNumTSOFields (Integer*4 TSOId, Integer*4 numFields)
- AskNumDataValues (Integer*4 TSOId, String:22 startTime, String:22 endTime, String:30 fieldNames[], Integer*4 numDataValues)
- AskTSOFieldsData (Integer*4 TSOId, String:22 startTime,String:22 endTime, String:30 fieldNames[], Integer*4 numFields[], String:variable fieldData[])
- AskTSOFieldsData1 (Integer*4 TSOId, String:22 startTime,String:22 endTime, String:30 fieldNames[], Integer*4 numFields[], String:variable fieldData[])
- AskStartKeyTime (Integer*4 TSOId, String:22 startTime)
- AskEndKeyTime (Integer*4 TSOId, String:22 endTime)

5.2 Translation of Binary Data to String

A TSO record logically consists of a key time followed by a sequence of heterogeneous fields. If the user requests an entire TSO record, the requested record is returned in a contiguous block of octets. For convenience this API includes calls that return the data as its logical Data Type. However since the data type of one field can be different from other fields, the API also includes calls where the system translates individual field's data into a common format (i.e. string) before returning the requested data. The following mapping guideline should be used for translating a binary data to a string for consistent implementation of the TSO:

Data Type	Translated String (left-justified)
1-byte, unsigned integer	3 bytes
1-byte, signed integer	4 bytes
2-byte, unsigned integer	5 bytes
2-byte, signed integer	6 bytes
4-byte, unsigned integer	10 bytes
4-byte, signed integer	11 bytes
4-byte, signed float (IEEE 754)	15 bytes
8-byte, signed float (IEEE 754)	25 bytes

As an example, if a field has two Integer*2 (signed) data values, say 1234 and 1244, then the returned value would be the 10 character string '1234 1244 '.

Throughout this API, when a key time is expressed as a string, it will be a 22 character string in the YYYYMMDDTHHMMSS.ssssss format where

YYYY	= year,
MM	= month,
DD	= day,
T	= the character"T" (separates date and time portions
HH	= hour,
MM	= minute,
SS	= second
.	= decimal point separating the integer and fractional portion of the seconds (may be replaced by a space character if the entire fractional portion of a second is replaced)
ssssss	= fractional portion of second (any number of the trailing digits may be replaced an equivalent number of space characters)

When TSO Record Fields are input or output in String format, they will use the format shown below.

Field	Size (octets)	Comments
Time offset	15 [base is Real*4]	In seconds. Left-justified. Blank padded to correct length.
Processing duration	15 [base is Real*4]	In seconds. Left-justified. Blank padded to correct length.
Time relationship	18	Contains one of the following values: 'start of duration ' 'middle of duration' 'end of duration ' Note that the returned value is padded with blanks if the length of the string is shorter than 18.
Data	Variable	Size depends on how the field

		data was declared and stored.

When TSO Records are input or output in String format, they will use the format shown below.

Field	Size (octets)	Comments
Key time	22	The key time
Field *x* *(Repeat once for each field)*	Variable 48+data	Use field data format shown above for each field

Appendix A: Example Responses to the Questions Asked in Section 2.

A.1 FITS Implementation Responses

How does FITS address the specific questions discussed for a time series object?

Here we discuss how they would be addressed within a single FITS HDU.   If we wish to consider a whole set of FITS files, it goes beyond the FITS standard.

2.2.1.1. What are available fields?
Look at TTYPEn entries in the headers.

2.2.1.2. What is the definition?
There is no requirement for a 'definition' of a column. Typically these are given as comments in the TTYPE fields.
Beyond this certain names are conventionally used.

2.2.1.3. What is the representation?
Look at TFORM, TDIM keywords.

2.2.1.4. What are the units?
Look at TUNIT keywords.

2.2.2.1 What is the meaning of the key time used in this object?
TIMEREF, TIMESYS, TIMEUNIT keywords (plus several others that are defined in HEASARC conventions).

2.2.2.2   What are the start/end times?
There are keywords available for specifying the minimum and maximum values of a column, however they are not required to be filled and rarely are.
Given the requirement that data be ordered in the time series, it is easy to read the first and last records since each record is of fixed length.

2.2.2.3 What is the processing duration for field 'x'?

In most cases this would be given in the nTDLTm keywords. However the FITS conventions primary define the separation between pixels. If there is 'dead-time' between observations (as in this reference model) there is no conventional representation of that in FITS.

2.2.2.4 What is the offset time for data field 'x'?

This is given by the nCRPXm and nCDLTm keywords.

2.2.2.5 How does the offset time relate to the processing duration?

See comments above.

2.2.2.6 Get the records in a given key interval?

This is an API issue, thus outside of FITS proper.

If efficiency is an issue, some kind of binary seach algorithm could be used, but FITS has no special support for indexing.

2.2.2.7 Get fields for a given key interval?

This is an API issue, thus outside of FITS proper. It should be straightforward to address.

2.2.3.1 Get fields within a specific interval?

This is an API issue. The implementation will need to carefully look at the various nXXXXm keywords.

2.2.3.2 How many records exist in an interval?

API issue.

2.2.3.3 What is the maximum record size?

If variable length columns are not used, then this is trivial. If variable length columns are used, then it is easy to search for the longest of these.

2.2.3.4 How many unique fields are present?
This may not be well posed. What does 'unique' mean in this context. E.g., are the 12 rates 12 unique fields, or just a single vector-valued field?

A.2 HDF4 Implementation Responses

Here are the HDF4 Answers to Questions to be Supported by the Time-series Object

2.2.1.1 What are available fields?
This info can be found by looking in the Vdata annotation, or by using standard HDF library function calls.

2.2.1.2 What is the definition of field "x"?
In HDF, there is no requirement for a 'definition' of a field, except to specify whether it is an int32 or a float64, etc. In this HDF4 TSO implementation, the definition is given in the the Vdata annotation and the field attributes.

2.2.1.3 What is the representation type for field "x"?
This info can be found by looking in the Vdata annotation, or by using standard HDF library function calls.

2.2.1.4. What are the units?
Found in the "Units" field attribute.

2.2.2.1 What is the meaning of the key time used in this object?
Found in the Vdata annotation, and in the "Units", "Time Offset", and "Time Relationship" field attributes of the fields "year", "day", "hour", etc.

2.2.2.2 What are the start/end times?
HDF does not require such information, but it could be an entry in the Vdata annotation.
Given the requirement that data be ordered in the time

series, it is easy to read the first and last records
since each record is of fixed length.

2.2.2.3 What is the processing duration for field 'x'?
Found in the "Processing Duration" field attribute.

2.2.2.4 What is the offset time for data field 'x'?
Found in the "Time Offset" field attribute.

2.2.2.5 How does the offset time relate to the processing duration?
Found in the "Time Relationship" field attribute.

2.2.2.6 Get the records in a given key interval?
This is an API issue, thus outside of the HDF domain. At the ASC, we define another Vdata within the data file which contains a map (index) relating record numbers to Time. This enables our user software to support this feature.

2.2.2.7 Get the values for fields 'x' and 'y' for the key time between time a and time b.
See 3.2.6 above. Extracting fields from given records is done using standard HDF library function calls.

2.2.3.1
Get the values for fields 'x' and 'y' when their individual times are between a and b.
This is an API issue. Additional software, calling the HDF 4 APIs, would be needed to return the data within the higher time resolution.

2.2.3.2 How many records exist between time a and time b?
API issue. See 3.2.6.

2.2.3.3 What is the maximum record size?
The records (Vdata instances) are of constant size. The size allocated for Vdata instances is obtained via a standard HDF library function call.

2.2.3.4 How many unique fields are present?

Not sure what this means. You can find out what fields are defined in a Vdata via standard HDF library function calls.

Appendix B: Example Implementations Detailed

B.1 CDF Implementation

The model data set is implemented using CDF's (Common Data Format) Variables and Attributes. Variable is an entity that represents/contains data, and attribute is the mechanism for storing metadata. There are two types of attributes in CDF: global and variable. Global attributes are used to describe a CDF file/data set (e.g. data set creator, file history, etc.) and to hold values that are common/global to the data set. Similarly, variable attributes are used to describe or provide additional information about variables. One or more variable attributes can be attached to a variable. For example, a variable can have MIN and MAX attributes to represent the minimum and maximum values allowed for this variable, and it can have a text description of the variable.

A CDF file can be created either by writing a program using CDF Application Prgramming Interfaces (APIs), creating a CDF skeleton table, or using the CDFedit interactive CDF editor. The easiest way to create a CDF file is by creating a CDF skeleton table that is an ASCII text file template in which one can define variables, atrributes, and other information such as file and variable compression methods to be used, text description of the CDF file/data set, data encoding scheme, and etc. SkeletonTable, one of the CDF tools distributed as part of the standard CDF distribution package, creates a skeleton table from an existing CDF file. Since the model data set is simple and only requires Variables and Attributes for implementation, a simple CDF file (test.cdf) that contains variables, attributes, and a few global atributes about the data set is selected (to minimize editing), and a skeleton table is generated using the following command at the operating system prompt:

```
skeletontable test.cdf
```

The above command produces a file called test.cdf.skt. The test.cdf.skt file is edited using an ASCII text editor to include the variables and attributes that are needed for the model data set. The edited skeleton table is then fed into the SkeletonCDF utility, another CDF tool distributed as part of the standard CDF distribution package, to generate the CDF file called TSO.cdf that contains the Time Series Object (TSO) model data set. The following command is used to generate the final CDF file:

```
skeletoncdf -cdf TSO.cdf test.cdf
```

The file extension of .ext is not required when specifying the input file name that is supplied to the SkeletonCDF utility to generate the TSO.cdf file.

Below is a skeleton table that implements the model data set with the following assumptions:

- Time offset doesn't vary from record to record.
- Processing duration doesn't vary from record to record.
- Time relationship doesn't vary from record to record.
- Key time and count rates vary from record to record.
- Fill data is used for missing data.

Variables KeyTime, SCLocation, CountRates, and Density represent the record key time, spacecraft location, count rate at each of 12 energy steps, and density that is determined from 12 count rates, respectively. Global attributes are used to describe the creator of the CDF file (TSO.cdf), to provide a brief description of this CDF file, and to further describe some variable attributes (e.g. gTimeOffset, gProcessingDuration, etc.). All variables except KeyTime are not independent or complete by themselves, and variable attributes are used to further describe these variables. The following mapping describes what variable attributes are used for each of the variables.

Variable Name	Attribute Name
SCLocation	Description
	Unit
	TimeOffset
	ProcessingDuration
	TimeRelationship
CountRate	Description
	Unit
	InitialTimeOffset
	TimeIncrement
	ProcessingDuration
	TimeRelationship
Density	Description
	Unit
	TimeOffset
	ProcessingDuration
	TimeRelationship

The variable attributes described above except for Description and Unit are also not complete by themselves and need additional information. For example, there's no information about what the units of ProcessingDuration is; there's no information about what the valid values are for the

TimeRelationShip attribute; and so on. The following global attributes are used to further describe these variable attributes:

Variable Attribute Name	Global Attribute Name
TimeOffset	gTimeOffset
InitialTimeOffset	gInitialTimeOffset
TimeIncrement	gTimeIncrement
ProcessingDuration	gProcessingDuration
TimeRelationship	gTimeRelationship

The logical CDF record, once the model is implemented, consists of the following fields: key time (KeyTime), spacecraft location (SCLocation), count rates(CountRates), density (Density). Note that the global and variable attributes are not part of the logical CDF record. With CDF, users can store and retrieve the entire, or a portion, of the logical CDF record at a time.

```
! Skeleton table for the "TSO.cdf" CDF.
! Generated: Friday, 19-May-2000 09:56:14
! CDF created/modified by CDF V2.7.0
! Skeleton table created by CDF V2.7.0b

#header

                   CDF NAME: TSO.cdf
              DATA ENCODING: IBMPC
                   MAJORITY: ROW
                     FORMAT: SINGLE

! Variables  G.Attributes  V.Attributes  Records  Dims  Sizes
! ---------  ------------  ------------  -------  ----  -----
   0/7           9             7          0/z      0


#GLOBALattributes

! Attribute            Entry   Data
! Name                 Number  Type       Value
! ---------            ------  ----       -----

  "Project"              1:    CDF_CHAR  {  "TSO Model Data in CDF" }  .

  "Author"               1:    CDF_CHAR  {  "David Han" }
                         2:    CDF_CHAR  {  "NASA/GSFC" }  .

  "ObjectTypes"          1.    CDF_CHAR  {  "TSO" }

  "TimeOrder"            1.    CDF_CHAR  {  "Increasing" }

  "gInitialTimeOffset"   1.    CDF_CHAR  {  "Initial time offset for variable
CountRate }
                         2.    CDF_CHAR  {  "UNIT = seconds" }
                         3.    CDF_CHAR  {  "ATTR = InitialTimeOffset" }
  "gTimeOffset"          1:    CDF_CHAR  {  "Time offset from key time"}
                         2:    CDF CHAR  {  "UNIT = Seconds"}
                         3:    CDF_CHAR  {  "ATTR = TimeOffset"}  .

  "gTimeIncrement"       1:    CDF_CHAR  {  "Time increment/offset for each
of the 12 count rates"}
                         2:    CDF CHAR  {  "UNIT = Seconds"}
                         3:    CDF_CHAR  {  "ATTR = TimeIncrement"}  .

  "gProcessingDuration"  1:    CDF_CHAR  {  "Processing duration time"}
                         2:    CDF CHAR  {  "UNIT = Seconds"}
                         3:    CDF_CHAR  {  "ATTR = ProcessDuration"}  .

  "gTimeRelationship"    1:    CDF_CHAR  {  "Valid time relationship values"}
                         2:    CDF CHAR  {  "ATTR = TimeRelationship"}
                         3:    CDF CHAR  {  "VALID = start of duration |"}
                         4:    CDF CHAR  {  "middle of duration |"}
                         5:    CDF_CHAR  {  "end of duration"}  .


#VARIABLEattributes

  "Description"
  "ProcessingDuration"
  "TimeIncrement"
  "TimeOffset"
  "InitialTimeOffset"
  "TimeRelationship"
  "Unit"



# zVariables

 ! Variable          Data      Number                    Record   Dimension
 ! Name              Type     Elements  Dims  Sizes  Variance  Variances
 ! --------          ----     --------  ----  -----  --------  ---------

"     "KeyTime          CDF_CHAR     15      1     1        T

     "Description"        CDF_CHAR   {  "Record key time, in UTC, whose value is
 stored as YYYYMMDDTHHMMSS where YYYY = year, MM = month, DD = day, T = separator
between date and time, HH = hour, MM = minute, and SS = second."}



 ! Variable          Data      Number                    Record   Dimension
 ! Name              Type     Elements  Dims  Sizes  Variance  Variances
 ! --------          ----     --------  ----  -----  --------  ---------

  "SCLocation"    CDF_FLOAT      1       1     3        T         T

   ! Attribute            Data
   ! Name                 Type       Value
   ! --------             ----       -----
```

```
    "Description"       CDF_CHAR  {  "X,Y,Z vector at time of key time" }
    "Unit"              CDF_CHAR  {  "Km" }
    "TimeOffset"        CDF_FLOAT {  0.0 }
    "ProcessingDuration" CDF_FLOAT {  0.0 }
    "TimeRelationship"  CDF_CHAR  {  "start of duration" }  .


! Variable          Data      Number                  Record   Dimension
! Name              Type     Elements  Dims  Sizes  Variance  Variances
! --------          ----     --------  ----  -----  --------  ---------

  "CountRate"     CDF_INT4       1       1     12       T         T

   ! Attribute           Data
   ! Name                Type     Value
   ! --------            ----     -----

    "Description"       CDF_CHAR { "Count rate at each of 12 energy steps.
Time offset for each of the 12 count rates is 5 seconds (whose value is
defined in the TimeIncrement attribute listed below) more than that for the
previous rate, where the first rate occurs at the key time and has a zero
offset (whose value is defined in the InitialTimeOffset attribute listed
below)."}
    "Unit"              CDF_CHAR {  "Number of particles per second" }  .
    "InitialTimeOffset" CDF_FLOAT {  0.0 }
    "TimeIncrement"     CDF_FLOAT {  5.0 }
    "ProcessingDuration" CDF_FLOAT {  0.0 }
    "TimeRelationship"  CDF_CHAR {  "start of duration" }



! Variable          Data      Number                  Record   Dimension
! Name              Type     Elements  Dims  Sizes  Variance  Variances
! --------          ----     --------  ----  -----  --------  ---------

  "Density"       CDF_FLOAT      1       1     1        T

   ! Attribute          Data
   ! Name               Type      Value
   ! --------           ----      -----

   "Description"       CDF_CHAR { "Density determined from 12 count rates"} .
   "Unit"              CDF_CHAR {  "Number of particles per cm3" }  .
   "TimeOffset"        CDF_FLOAT {  0.0 }
   "ProcessingDuration" CDF_FLOAT {  55.0 }
   "TimeRelationship"  CDF_CHAR {  "start of duration" }  .
  ! (Note:  It would also be correct to use a time offset value of 27.5
  ! with a time relationship of 'middle of duration')

#end
```

NOTE:

1. The "Record Variance" column specifies whether or not the variable's values change from record to record.

2. The "Dimension Variances" column specifies whether or not the values change along the corresponding dimension.

3. The "Record Variance" and "Dimension Variances" columns can have one of the following values:

T - True.
F - False
' ' - Blank value. This means "not applicable."

B.2 HDF4 Implementation

We use two HDF building blocks to implement the TSO in HDF - Vgroups and Vdatas. Vgroups are generic grouping elements allowing a user to associate related objects within an HDF file. As Vgroups can contain other Vgroups, it is possible to build a hierarchical file. Vdatas are generic list objects. Data is organized into "fields" within each Vdata. Each field is identified by a unique "field name". The type of each field may be any of the basic types that HDF supports. Fields of different types may exist within the same Vdata.

For this example, Vgroups are not very important. If there were data from several different instruments within the data file, one could define a Vgroup for each instrument, for example. In any case, within a Vgroup we can define a Vdata, and within the Vdata we can define the fields into which the data is to be organized. So, there will be a correspondence between the Vdata fields and the Words in the model data set.

We use HDF File Annotations and HDF Object Annotations and HDF Attributes to record metadata within the HDF files. Any generic HDF data browser tool should be able to display all annotations and attributes contained within any HDF file.

HDF annotations are basically containers into which one can dump short text descriptions. File annotations are used to record global information about the dataset.
Object annotations can be attached to Vgroups or Vdatas or other objects within a HDF file. We would attach an object annotation to the TSO Vdata that would describe the Vdata fields.

HDF attributes each have a name, a data type, a number of attribute values, and the attribute values themselves. Any number of attributes can be assigned to either a Vdata or any field within a Vdata, as long as they are named uniquely.

We would use HDF attributes to define units for the Vdata fields, offsets for the count-rates, MAX/MIN limits, etc.

An easy way to define the Vdata fields is via a C data structure. Given this definition, a straightforward series of calls to HDF library routines is needed to set up the Vdata, and to begin reading/writing data records from/to a data file. This definition (including the comments) can become a part of the data file if it is later incorporated into an Annotation of the Vdata (see below).

```
struct TOS_data_1min {

  /* UT time-tag data */
  int32   year;              /* integer year */
  int32   day;               /* integer day of year */
  int16   hr;                /* hour of day */
  int16   min;               /* min of hour */
  float64 fp_year;           /* floating point year */
  float64 fp_doy;            /* floating point Day of Year */
  float64 ACEepoch;          /* Number of seconds since 00:00:00 01-01-1996 UT*/

  float32 sc_locationX;      /* X-component of S/C loc. at time tag (GSE) */
  float32 sc_locationY;      /* Y-component of S/C loc. at time tag (GSE) */
  float32 sc_locationZ;      /* Z-component of S/C loc. at time tag (GSE) */

  int32 count_rate1;         /* count rates at each of 12 energy steps. */
  int32 count_rate2;         /* Each rate is offset +5secs from the previous */
  int32 count_rate3;         /* rate, with the first rate offset 0 secs from */
  int32 count_rate4;         /* the time tag. If we knew the energy steps, we */
  int32 count_rate5;         /* might mention them here */
  int32 count_rate6;
  int32 count_rate7;
  int32 count_rate8;
  int32 count_rate9;
  int32 count_rate10;
  int32 count_rate11;
  int32 count_rate12;

  float32 density;           /* determined from the 12 rates by taking */
                             /* moments of distribution function. Time tag
                             /* designates the start-time of the processing
                             /* interval for this item */
};
```

Note that the time definition above contains more fields than the model data mandated - that's just to indicate that there are many ways of specifying Time, some useful for plotting routines, some for readability, etc, etc. No matter which subset one decides upon, one is going to annoy somebody...

Note also that the S/C location and count-rates could easily be defined as arrays. However, breaking them out into scalar quantities makes it easier to label each item with its own metadata (see below).

In this implementation, all records will have the same length. Data gaps can either be filled with records containing fill-data, or not...

Metadata (Annotations and Attributes)

Examples of file annotations that might be added to the HDF file are:

· Data description: A general description of the data, including processing dates and version numbers. One would define how gappy data is handled here.
· Contact Info: Instrument team contact information.
· Release Notes: Provided by the instrument team.

The C data structure defined above (with comments) would be a minimal starting point for
An object annotation that would be attached to the TSO Vdata.

One use of HDF attributes would to define units for the Vdata fields. For instance, the "min" field could have an attribute with name="UNITS", data-type= char, num_of_values=7, and values="minutes". Another use of attributes would be to define the offsets for the count-rates. The table below shows examples of attributes we could attach to each of the fields in the TSO Vdata:

	Attributes			
Field Name	Units	Processing Duration	Time Offset	Time Relationship
year	"years"	"NA"	"0 sec"	"Begin of Duration"
day	"days"	"NA"	"0 sec"	"Begin of Duration"
hr	"hours"	"NA"	"0 sec"	"Begin of Duration"
min	"minutes"	"NA"	"0 sec"	"Begin of Duration"
fp_year	"years"	"NA"	"0 sec"	"Begin of Duration"
fp_doy	"days"	"NA"	"0 sec"	"Begin of Duration"

ACEepoch	"seconds"	"NA"	"0 sec"	"Begin of Duration"
sc_locationX	"km"	"0 sec"	"0 sec"	"Begin of Duration"
sc_locationY	"km"	"0 sec"	"0 sec"	"Begin of Duration"
sc_locationZ	"km"	"0 sec"	"0 sec"	"Begin of Duration"
count_rate1	"counts/second"	"0 sec"	"0 sec"	"Begin of Duration"
count_rate2	"counts/second"	"0 sec"	"5 sec"	"Begin of Duration"
count_rate3	"counts/second"	"0 sec"	"10 sec"	"Begin of Duration"
count_rate4	"counts/second"	"0 sec"	"15 sec"	"Begin of Duration"
count_rate5	"counts/second"	"0 sec"	"20 sec"	"Begin of Duration"
count_rate6	"counts/second"	"0 sec"	"25 sec"	"Begin of Duration"
count_rate7	"counts/second"	"0 sec"	"30 sec"	"Begin of Duration"
count_rate8	"counts/second"	"0 sec"	"35 sec"	"Begin of Duration"
count_rate9	"counts/second"	"0 sec"	"40 sec"	"Begin of Duration"
count_rate10	"counts/second"	"0 sec"	"45 sec"	"Begin of Duration"
count_rate11	"counts/second"	"0 sec"	"50 sec"	"Begin of Duration"
count_rate12	"counts/second"	"0 sec"	"55 sec"	"Begin of Duration"
density	"particles/cm3"	"55 sec"	"0 sec"	"Begin of Duration"

Note that all the attributes above are implemented as character strings, which limits their usefulness in computations. An alternative might be to implement the offsets as integers, for example.

However, then one must find another way to communicate the units of the offsets, possibly as attributes attached to the Vdata (i.e. one level up in the hierarchy). Such schemes require a smart API to make proper use of them...

Some fields may need more attributes (eg. MIN and MAX limits for plotting, or count rate energy steps). That's OK, HDF allows for different fields to have different numbers of attributes.

B.3 IDFS Implementation

There are various ways in which the model data set can be stored under IDFS, which is probably a statement that can be made by other formats as well. Under IDFS, you store the RAW data and the procedures to convert that raw data into geophysical quantities. That is not to say that computed data cannot be stored under IDFS; it is simply that the strength of IDFS is in the ability to modify the procedure for data conversion into physical units without the need to re-process all data sets since the raw data is what is stored.

With IDFS, data is grouped into virtual (or logical) instruments. A virtual instrument is a group of sensors that are linked together by commonality; therefore, it makes sense to treat the sensors together. A virtual instrument can be classified as either a scalar instrument or a vector instrument. A scalar instrument returns singular data quantities that are dependent only upon time and position. An IDFS vector instrument returns one-dimensional data quantities that have a functional dependence upon a single variable called the scan variable. The length of this 1-D vector is virtual instrument dependent.

There appears to be three different types of data contained in the model data set: (1) the s/c location (which are three scalar values), (2) the count rates consisting of 12 elements, and (3) density (a scalar processed value). It is assumed that the spacecraft location and count rates are measurements with discrete values. These data values should be stored in their raw form and the IDFS should contain the necessary descriptions for converting the stored data numbers into geophysical numbers. If the assumption is not correct and these are all computed quantities, they can be stored as is within IDFS as data.

One possible scheme would be to subdivide the data into 3 virtual instruments, which are referred to as (1) POSITION, (2) COUNT_RATE, and (3) DENSITY. In the IDFS paradigm, each data set is written into two types of files: the header file (H) and the data file (D). File names link the proper header and data files together. There is also a VIDF file and a PIDF file per virtual instrument, as described later. Using the prescription as laid out, one year of data at one day per file would result in the following maximum number of files:

- IDFS data files = 3 virtual instruments * 365 days in year = 1095
- IDFS header files = 3 virtual instruments * 365 days in year = 1095
- IDFS VIDF files = 3 virtual instruments = 3
- IDFS PIDF files = 3 virtual instruments = 3

Total = 1095 + 1095 + 3 + 3 = 2196 files

An alternative approach would be to create IDFS files that are one year in duration, thereby resulting in the following maximum number of files:

- IDFS data files = 3 virtual instruments = 3
- IDFS header files = 3 virtual instruments = 3
- IDFS VIDF files = 3 virtual instruments = 3
- IDFS PIDF files = 3 virtual instruments = 3

Total = 3 + 3 + 3 + 3 = 12 files

The latter approach results in a fewer number of files. This is beneficial when data is being promoted from an IDFS archival site; that is, there would be a smaller number of files to be transferred to the local machine that is requesting the IDFS data. In addition, there would be fewer entries for the IDFS database. This results in a smaller number of comparisons being made when the database is being searched for a particular time period.

The Virtual Instrument Description File (VIDF) is a complete description of the virtual instrument. The VIDF file is meant to be easily updated and to contain all of the data that may be periodically updated due to either refinement in the instrument calibration or due to the degradation within the instrument. There must be at least one VIDF file defined for each IDFS virtual instrument. If data within the VIDF changes with time, for example calibration coefficients, additional VIDF files can be defined. The VIDF file provides a general description of the measurements being stored in IDFS format, and contains the data reconstruction parameters that are needed in order to transform the raw data into physical units. In addition, the VIDF file provides information by which data from the data file can be extracted, such as:

1. the field DATA_LEN holds the size of the data records contained in the data file
2. the field MAX_NSS defines the maximum number of sensor sets defined in each data record
3. the field TDW_LEN defines the word length (bits) for each sensor contained within each data record

4. the field FILL contains the designated fill data value used in the IDFS data records to indicate missing or fill data

The Plot Interface Definition File (PIDF) is an optional file under the IDFS format. The word optional in this context means that it is not necessary in the use of any of the IDFS data access software; that is, data in IDFS format can be located through the database, accessed and converted to units without the PIDF file. If, however, you plan to use any IDFS based data display or analysis software that has been developed by Southwest Research Institute (SwRI), the PIDF file is required.

The PIDF is best described as an interface file between the IDFS VIDF definitions and a general user interface to a display or analysis program, providing a large number of display attributes. The PIDF file is an ASCII file that describes how to display the data in a meaningful way. The PIDF file defines the units that are available for each sensor including the correct VIDF tables to apply, how they are applied and the scaling limits (min/max) for each set of units.

The header files contain data which, for the most part, is slowly varying in time and need not be repeated every data record. Each IDFS data record points to a header record that describes the state of the instrument at that particular point in time. The format of the header record is shown below in the form of a C data structure, utilizing user-defined data types to address the issue of porting the source code to multiple platforms:

```
struct
{
   SDDAS_SHORT     hdr_len;                      /* header record size, in bytes
   SDDAS_SHORT     year;                         /* 4-digit year
   SDDAS_SHORT     day;                          /* Julian day of year
   SDDAS_CHAR      time_units;                   /* the scaling factor to convert
                                                 /* DATA_ACCUM value to seconds
   SDDAS_UCHAR     i_mode;                       /* no. of status bytes defined
   SDDAS_LONG      data_accum;                   /* time to acquire a single
                                                 /* data measurement
   SDDAS_LONG      data_lat;                     /* dead time in microseconds
                                                 /* between successive data
                                                 /* acquisitions
   SDDAS_LONG      swp_reset;                    /* dead time required to restart
                                                 /* data sampling of each sensor
   SDDAS_LONG      sen_reset;                    /* dead time required to restart
                                                 /* data sampling for the entire
                                                 /* instrument
   SDDAS_SHORT     n_sen;                        /* no. of sensors within the
                                                 /* data record
   SDDAS_USHORT    n_sample;                     /* no. of data samples being
                                                 /* returned for each sensor
   SDDAS_SHORT     scan_index[1 or n_sample];    /* offsets used to index into
                                                 /* any VIDF tables which are a
```

```
                                                            /*  function of scan
    SDDAS_SHORT      sensor_index[n_sen];                   /*  the sensor numbers in the
                                                            /*  order that they are written
                                                            /*  in the data record
    SDDAS_UCHAR      d_qual[n_sen];                         /*  data quality flags
    SDDAS_UCHAR      mode_index[i_mode];                    /*  the current state of each
                                                            /*  defined status byte
};
```

D_qual is an index into an array of data quality flags defined in the VIDF file. At a minimum, each VIDF file should define 2 levels of data quality, good and bad data. The fields **data_accum**, **time_units**, and **data_lat**, taken together, define the total time between successive accumulations. Note that for many scalar data sets, the accumulation time (**data_accum**) is set to zero to indicate that the accumulation is instantaneous. In these cases, the **data_lat** field is used to give the time between successive measurements. For those virtual instruments that are classified as vector instruments, every sample within the sweep has a finite acquisition time, regardless of how small. It is important to keep the duration of the window during which the acquisition took place; therefore, **data_accum** should not be set to zero. In addition, the **swp_reset** field is needed to define the dead time that is needed to reset themselves between successive sweeps. For many particle instruments, this is equivalent to the flyback time.

The vast majority of all of the telemetered data is stored within the data file, which contains the most rapidly varying data. The data records contain the base time tag for the data and raw, unprocessed binary data. Unlike the header record, the data record does not vary in size. The size is specified in the VIDF file. The format of the data record is shown below in the form of a C data structure, utilizing user-defined data types to address the issue of porting the source code to multiple platforms:

```
struct
{
    SDDAS_LONG       dr_time;                   /*  start time for the 1st data element
                                                /*  of the first sensor set (msec)
    SDDAS_LONG       spin;                      /*  azimuthal rate of rotation in
                                                /*  msec/rev
    SDDAS_LONG       sun_sen;                   /*  time of day (msec) at which a
                                                /*  predefined location crosses the
                                                /*  azimuthal zero degree position
    SDDAS_LONG       hdr_off[max_nss];          /*  byte offset into header file, one
                                                /*  per sensor set
    SDDAS_LONG       nss;                       /*  no. sensor sets within data record
    SDDAS_UCHAR      data_array[data_size];     /*  all of the data stored within the
                                                /*  data record
};
```

Note that the **data_array** is generically assigned the data type SDDAS_UCHAR, which is an unsigned character (8 bits). The data may be stored within the field with a base length of 8, 16 or

32 bits. The storage boundary used for individual data within a particular data file is determined from the VIDF file and is used by the IDFS data access software to correctly unpack the data.

Data storage in the IDFS data record is organized along the concept of sensor (primary) data, calibration (secondary) data and sensor sets. Sensor data is the basic, primary measurement and is placed in the **data_array** field first. Calibration data is ancillary data that is necessary to interpret the primary data (e.g., automatic gain correction values). Not all virtual instruments have calibration data defined. Calibration data is placed in the **data_array** field after all sensor data has been written. These two data matrices (sensor and calibration) taken collectively are what is referred to as an IDFS sensor set. A different header record can describe each sensor set since there is one **hdr_off** value defined per sensor set.

For the TSO model data set, each data record represents one minute of data. Since the TSO model data set contains a small amount of data, under IDFS, the data could be written in hour records, or 60 sensor sets per record. Or better yet, the entire day of data could be packed into one record of 3600 sensor sets.

How gaps are incorporated into the IDFS data files depends upon how the data is structured in the data records. If a single sensor set is written per data record, gaps are handled by simply omitting the missing data records. The timetags for the data that are written will reflect the gap in time. If multiple sensor sets are written per data record, a fill value to be inserted as a place holder for bad or missing values should be defined in the VIDF file. This fill value will need to be inserted into the appropriate part of the data.

For IDFS, the data types have the following meaning:

SDDAS_INT	4-byte signed integer
SDDAS_LONG	4-byte signed integer
SDDAS_2LONGS	8-byte signed integer
SDDAS_FLOAT	4-byte floating point
SDDAS_DOUBLE	8-byte floating point
SDDAS_SHORT	2-byte signed integer
SDDAS_CHAR	1-byte signed integer or single character
SDDAS_UINT	4-byte unsigned integer
SDDAS_ULONG	4-byte unsigned integer
SDDAS_USHORT	2-byte unsigned integer
SDDAS_UCHAR	1-byte unsigned integer

For word 1, the time tag in the format, YYYYDDDtHHMM, the value would be broken down into 3 values: (1) 4 digit year value, (2) julian day of year value (1 to 365 (366)), (3) time of day in millisecond resolution. The first two quantities would be placed in the header record associated with each of the 3 virtual instruments. The third value would be placed in the data record associated with each of the 3 virtual instruments and would be computed using the following algorithm:

(HH * 3600 + MM * 60) * 1000

since the base time tag in the data record (**dr_time**) is expressed in milliseconds.

Word 2, s/c location, would only be placed into the data record for the POSITION virtual instrument. The location values are assumed to be instantaneous values determined at the time tag, with all three components determined simultaneously. The timing information stored in the header record for these data values would describe a data accumulation value of zero and a data latency value of 1 minute or 60 seconds (this is the time between successive data samples). The VIDF file would describe 3 scalar sensors taken in parallel, each being a floating-point real number. Since there isn't that much s/c location data, the data could be repackaged into one hour or one day data records instead of one-minute data records. It is always assumed that the time of day value written in the data file is the start of the accumulation time of the first element of data in a data record.

An example header record for the POSITION virtual instrument would be:

```
struct
{
   SDDAS_SHORT     hdr_len;
   SDDAS_SHORT     year = word 1 time tag component YYYY;
   SDDAS_SHORT     day = word 1 time tag component DDD;
   SDDAS_CHAR      time_units = 0;
   SDDAS_UCHAR     i_mode = 0;
   SDDAS_LONG      data_accum = 0;
   SDDAS_LONG      data_lat = 60000000;
   SDDAS_LONG      swp_reset = 0;
   SDDAS_LONG      sen_reset = 0;
   SDDAS_SHORT     n_sen = 3;
   SDDAS_USHORT    n_sample = 1;
   SDDAS_SHORT     scan_index[1] = {0};
   SDDAS_SHORT     sensor_index[3] = {0, 1, 2};
   SDDAS_UCHAR     d_qual[3] = {1, 1, 1};
};
```

Note that when the header record is written, the value for **hdr_len** should be computed using a sizeof function, in which case, the value may be rounded to the nearest word size. In the example above, the total size in bytes (**hdr_len**) tallies in at 39.

An example data record that represents one minute of data per record for the POSITION virtual instrument would be:

```
struct
```

```
{
   SDDAS_LONG        dr_time =  (HH *  3600 + MM *  60) *  1000;
   SDDAS_LONG        spin = 0;
   SDDAS_LONG        sun_sen = 0;
   SDDAS_LONG        hdr_off[1]  = {0};
   SDDAS_LONG        nss = 1;
   SDDAS_UCHAR       data_array[3]  = {x, y, z};
};
```

An example data record that represents one hour of data per record for the POSITION virtual instrument would be:

```
struct
{
   SDDAS_LONG        dr_time =  (HH *  3600 + MM *  60) *  1000;
   SDDAS_LONG        spin = 0;
   SDDAS_LONG        sun_sen = 0;
   SDDAS_LONG        hdr_off[60]  = {0, 0, 0, ..., 0};
   SDDAS_LONG        nss = 60;
   SDDAS_UCHAR       data_array[180]  = {x, y, z, x, y, z, ...., x, y, z};
};
```

Note that the **hdr_off** values in the data record shown above are all the same. At a minimum, this implies that all samples were taken on the same year / day of year combination. If a day boundary is crossed, the **hdr_off** values should be modified to point to the header record that contains the correct year / day of year values. Since the data records are being bundled on an hour boundary, this would not happen unless time started somewhere within the hour instead of at the beginning of the hour.

Words 3 - 14, count rates, would only be placed into the data record for the COUNT_RATE virtual instrument. For this data set, an assumption that all twelve measurements are a function of a single parameter, that being energy, is made. This assumption is based on the wording "1st of 12 energy steps". Based on this assumption, the data would describe a single sensor coming from an IDFS "vector" instrument. A vector instrument is an instrument whose sensors, or data products, represent multivalue (1-D) data sets that have known functional dependencies other than time or position, i.e., a particle spectrometer which returns counts as a function of energy. The length of this 1-D array would be defined as 12 for this virtual instrument. What are also needed for the description of this data set are what energies these count rates depend on. In order to provide this information, twelve index values, labeled 0 through 11, will be placed in the header record within the header file. These twelve indexes will provide the ability to link each count rate to the energy that the count rate is dependent upon. In the VIDF file, the data will be described as 4-byte integers and there are 12 elements in a vector scan. Note that if you wanted you could just store the count rather than the rate and let the IDFS compute the rate. If this were true, in the VIDF file, a table could be specified to generate the energy value given the sequence number. It is assumed that the time of day value written in the data file is the start of the accumulation time of the first element of data in a data record.

An example header record for the COUNT_RATE virtual instrument would be:

```
 struct
{
   SDDAS_SHORT     hdr_len;
   SDDAS_SHORT     year = word 1 time tag component YYYY;
   SDDAS_SHORT     day = word 1 time tag component DDD;
   SDDAS_CHAR      time_units = -6;
   SDDAS_UCHAR     i_mode = 0;
   SDDAS_LONG      data_accum = 1;
   SDDAS_LONG      data_lat = 4999999;
   SDDAS_LONG      swp_reset = 0;
   SDDAS_LONG      sen_reset = 0;
   SDDAS_SHORT     n_sen = 1;
   SDDAS_USHORT    n_sample = 12;
   SDDAS_SHORT     scan_index[12] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
   SDDAS_SHORT     sensor_index[1] = {0};
   SDDAS_UCHAR     d_qual[1] = {1};
};
```

This data has an accumulation period, which is not known, and a data latency, which is not known, only that the sum of the accumulation and latency is 5 sec. In the example header record shown above, the 5 seconds is split between the data accumulation value, which is specified in seconds, and the data latency value, which is specified in microseconds. If all 12 count rate values are not returned, a new header could be created and the header record would specify the actual number of values returned and the indexes (energy step) which correspond to the values that were returned. For example, if 6 values were returned, that being every other value, six indexes would be specified {0, 2, 4, 6, 8, 10}.

If the 12 values are not functions of a single parameter, but rather 12 distinct measurements, then the data would describe twelve IDFS sensors coming from an IDFS "scalar" instrument. A scalar instrument is an instrument whose sensors, or data products, represent a set of singular data values that are dependent only upon time and position, e.g. a housekeeping temperature monitor. The VIDF would describe twelve scalars taken in parallel, and the time offsets (0, 5, ... 55) would be placed in the VIDF file and utilized when the time tag for each individual sensor is computed.

An example data record that represents one minute of data per record for the COUNT_RATE virtual instrument would be:

```
 struct
{
   SDDAS_LONG      dr_time = (HH * 3600 + MM * 60) * 1000;
   SDDAS_LONG        spin = 0;
   SDDAS_LONG        sun_sen = 0;
   SDDAS_LONG        hdr_off[1] = {0};
   SDDAS_LONG        nss = 1;
   SDDAS_UCHAR       data_array[12] = {rate1, rate2, ... rate12};
};
```

An example data record that represents one hour of data per record for the COUNT_RATE virtual instrument would be:

```
struct
{
   SDDAS_LONG        dr_time = (HH * 3600 + MM * 60) * 1000;
   SDDAS_LONG        spin = 0;
   SDDAS_LONG        sun_sen = 0;
   SDDAS_LONG        hdr_off[60] = {0, 0, 0, ..., 0};
   SDDAS_LONG        nss = 60;
   SDDAS_UCHAR       data_array[720] = {rate1, rate2, ... rate12,
                                       rate1, rate2, ... rate12,
                                       rate1, rate2, ... rate12,
                                       ...,
                                       rate1, rate2, ... rate12};
};
```

Word 15, density, would only be placed into the data record for the DENSITY virtual instrument. The DENSITY header file has one sensor defined with a data accumulation value of 55 seconds and a data latency value of zero seconds. The VIDF file defines one sensor that is a real value that is 4-bytes long. Since the density is highly processed data, this is probably the best way to store this data. The 4-byte values are probably not easily converted into something that can pack better. When generating the density data time tag, the time tag is the start of the data accumulation time. Writing only one data set per data record is a poor way to save the density data. First, assume that if there is missing count rate data, then there is no density data. Since there is one value per data record, you could just not write that record, so there will be a gap that could be handled by IDFS. However, the recommendation for this data set would be to write the density in hour records, or 60 values per data record. A fill value could be defined which could be put in place of bad or missing values. Another alternative is to pack the entire day of data into one record of 3600 elements.

An example header record for the DENSITY virtual instrument would be:

```
struct
{
   SDDAS_SHORT       hdr_len;
   SDDAS_SHORT       year = word 1 time tag component YYYY;
   SDDAS_SHORT       day = word 1 time tag component DDD;
   SDDAS_CHAR        time_units = 0;
   SDDAS_UCHAR       i_mode = 0;
   SDDAS_LONG        data_accum = 55;
   SDDAS_LONG        data_lat = 0;
   SDDAS_LONG       swp_reset = 0;
   SDDAS_LONG       sen_reset = 0;
   SDDAS_SHORT      n_sen = 1;
   SDDAS_USHORT     n_sample = 1;
   SDDAS_SHORT      scan_index[1] = {0};
   SDDAS_SHORT      sensor_index[1] = {0};
   SDDAS_UCHAR      d_qual[1] = {1};
};
```

An example data record that represents one minute of data per record for the DENSITY virtual instrument would be:

```
 struct
{
   SDDAS_LONG       dr_time = (HH * 3600 + MM * 60) * 1000;
   SDDAS_LONG       spin = 0;
   SDDAS_LONG       sun_sen = 0;
   SDDAS_LONG       hdr_off[1] = {0};
   SDDAS_LONG       nss = 1;
   SDDAS_UCHAR      data_array[1] = {density};
};
```

An example data record that represents one hour of data per record for the DENSITY virtual instrument would be:

```
 struct
{
   SDDAS_LONG       dr_time = (HH * 3600 + MM * 60) * 1000;
   SDDAS_LONG       spin = 0;
   SDDAS_LONG       sun_sen = 0;
   SDDAS_LONG       hdr_off[60] = {0, 0, 0, ..., 0};
   SDDAS_LONG       nss = 60;
   SDDAS_UCHAR      data_array[60] = {density, density, density, ..., density};
};
```

B.4 FITS Implementation

The FITS data format is defined solely in terms of the data structure as stored on disk or tape media. No standard FITS API exists although the format is nearly universally supported in astronomy analysis software.

A FITS file consists of one or more Header-Data Units (HDUS). Descriptive information about the data is placed in the header which completely defines the structure of the data that follows. The header may optionally contain information on the semantics of the data, i.e., the unit, definitions of the axes and so forth.

FITS addresses the relationships between the numbers in the FITS files and the physical units in several ways. For some keywords values, e.g., coordinates, the units are defined in the FITS standard. For other keywords the units are often given using comments. The relationship between data elements and phyical units is more completely detailed using the FITS WCS conventions which have now been submitted for formal ratification by appropriate national committees. For scalar elements of tables, there are explicit scale, offset and unit keywords. For non-scalars -- either in tables or images -- FITS allows the user to associate each dimension (viewed as pixels) of the object with a physical axis where appropriate keywords describe a linear transformation from the pixel offsets to physical units.

Frequently a scalar value is treated as being a vector of length 1, or a non-scalar is described as having higher dimensionality than is required with the extra dimensions all have length 1. These 'extra' dimensions allow the user to specify additional information about the physical characteristics of the data. E.g., an two dimensional image of the sky may be described as being NxNx1 rather than just NxN. The third dimension can be the energy dimension and allows the FITS writer to describe the energy range of the observation. Similarly a rates array may be an Nx1 array rather than just N, with the additional axis again describing the energy. This could be taken even further. A rate array might be described as Nx1x1x1 where the keywords describing the first two additional axes give the position of the observation and then the energy.

Some of the standards for specification of this metadata have been standardized in protocols approved by the International Astronomical Union. Others are specified in more localized protocols. In the example below we specify some metadata using protocols of the High Energy Astronomy Science Archive Research Center (HEASARC). While these protocols have been widely adopted in some areas of astronomy, they are not universally accepted.

For the reference data set, an implementation as a FITS table, or a set of tables is most natural. We have implemented the reference data set using the same file structure as in the reference model, with each day kept as a separate file. It would be perfectly feasible to organize all of the data in a single file, or to have multiple HDU's with different days within a single FITS file.

To realize a FITS file with these data we would need to create a file with two HDU's. The first is a dummy HDU where the header can contain descriptive information about the file, but will contain no actual data. Table data is not permitted in the first HDU. The header for an HDU comprises a series of 80 byte records terminated by an END record. The dummy HDU header might look like:

```
SIMPLE  =                    T  / Indicates this file is FITS
BITPIX  =                    8  / Byte data (but meaningless in this context)
NAXIS   =                    0  / Dimensionality of data -- there isn't any
EXTEND  =                    T  / There may be additional HDU's
```

```
END
```

The meaning of each line is given after the '/' which begins the comment field of a FITS header line.

Following the primary HDU (with some padding required by rules originally designed to enable FITS to work well with tape media) the header for the second HDU is written. This consists of a series of required keywords which indicate the type and size of the table, and then definitions of each of the columns. Only the actual data format of the column is required but much other information about the column can be given. An example header which describes the reference data set is given below. Comments describe each line in the header.

```
XTENSION= 'BINTABLE'           / FITS binary table
BITPIX  =                    8 / Mandatory value for binary tables
NAXIS   =                    2 / Mandatory value for binary tables
NAXIS1  =                   72 / Number of bytes per row
NAXIS2  =                 1440 / Number of rows (assuming no missing data)
PCOUNT  =                    0 / No variable length data
GCOUNT  =                    1 / Mandatory value
TFIELDS =                    4 / Number of distinct fields in table

EXTNAME = 'RATE      '           / A descriptive name of the extension
HDUCLASS= 'OGIP      '         / Conforms to OGIP/HEASARC standards.
HDUCLAS1= 'LIGHTCURVE'           / HEASARC type hierarchy
HDUCLAS2= 'RAW       '           /    "
HDUCLAS3= 'RATE      '           /    "

TTYPE1  = 'TZERO    '            / In seconds since 1980.
TTYPE2  = 'POSITION'             / Spacecraft position at time
TTYPE3  = 'RATES    '            / Rate array
TTYPE4  = 'DENSITY '             / Averaged rate array.

TFORM1  = 'D        '            / A double precision scalar
TFORM2  = '3E        '           / A vector of three reals
TFORM3  = '12J      '            / A vector of 12 4 byte integers
TFORM4  = '1E        '           / A single precision scalar
                                   However we will treat it
                          as a one element vector
                          to allow us to describe
                          its temporal coverage.

TUNIT1  = 's        '            / Time in seconds
TUNIT2  = 'km       '            / Kilometers from solar barycenter.
TUNIT3  = 'count    '
TUNIT4  = 'count/s '

TDIMS3  = '(12,1)   '            / Treat the rates as a 12x1 two-dimensional
                                   array.

3CTYP1  = 'TIME     '            / The data in the
```

```
                                    third column is a function of time.
                                The trailing '1' is needed since this
                                data might be multidimensional, i.e.,
                                might have multiple axes.  And we have
                                added a dummy dimension to the
                                data to enable us to specify
                                the energy range of the data.


3CRVL1  = 'TZERO    '               / The reference time for a given row of the
                                      data is in the 'TZERO' field.

3CDLT1  =                         5 / Size of pixels

3CRPX1  =                       0.5 / The first pixel in a FITS array includes
                                      the range of pixel values 0.5 to 1.5.  The
                                reference time specified in 2CRVL1 refers
                                to the beginning of the first pixel.

3CTYP2  = 'ENERGY   '               / The second axes is in energy space.
3CRVL2  =                      20.0 / The reference energy is 20 keV
3CRPX2  =                         1 / The reference energy is in the middle of
                                      the first pixel
3CDLT2  =                       5.0 / The width of the pixels in energy space
                                      is 5 keV.

4CTYP1  = 'TIME     '               / This column is also to be treated
                                      as having 'pixel's in time.  There
                                will only be a single pixel in
                                each row though.

4CRVL1 = 'TZERO     '               / The reference time is given in the TZERO
                                      column

4CDLT1 =                         55 / The pixels are 55 seconds long.

4CRPX1 =                        0.5 / The reference time refers to the beginning
                                      of the first (and in this case only) pixel

TIMEUNIT= 's        '               / Unit used in time information (HEASARC)

TIMESYS = 'UTC      '
TIMEREF = 'HELIOCENTRIC'            / Reference location for timing (HEASARC)
TIERABSO=                      0.34 / Seconds (Absolute timing accuracy) (HEASARC)

....
END
```

After a small amount of padding the data would be written in big-endian, IEEE standard format.

Note the extensive use of the nTYPm, nCRVLm, nCRPXm and nCDLTm keywords. These provide quite a flexible approach to describing where a 'pixel' of data exists in FITS space. The first number refers to the column being described. Each column can have a n-dimensional vector as a value. These keywords indicate what the axes of these vectors are and how the user can find the value along these axes appropriate for a given pixel. The third and fourth columns both describe pixels in space and thus we give them a time axis. Note that we can treat a scalar as a single element vector along some axis. We do this to describe the time interval associated with the fourth column. We also do it to describe the range associated with the rates in column 3.

There is considerable work in progress in standardizing these associations in the FITS community. Time and projected positions are described using agreed and relatively sophisticated protocols. Work is still underway in describing energy-based (e.g, frequency, wavelength) columns.

B.5 PDS Labels Implementation

Overview

The PDS label architecture is oriented to providing an archival description of a data file, rather than to supporting display or manipulation of the contents. The PDS SERIES object which is used in this example is a slight specialization of the PDS TABLE object, which is just a flat file of ASCII or binary values (depending on the value of the INTERCHANGE_FORMAT keyword).

TSO implementation.

Offsets. These are normally specified in descriptive text. The sampling parameter keywords can also be used to indicate a repeating group of regular observations as illustrated in the example.

File organization. The organization of data into physical files in the PDS architecture is done using a set of PDS guidelines based on size and time. In this case each physical file is specified to be one day's worth of data records or 1440 records. There would then be conventions for directory names and file names that would provide convenient access to a file containing a certain period of time.

Units. Units can be specified for a column object using the UNIT keyword. When units need to be associated with a sampling parameter, the SAMPLING_PARAMETER_UNIT keyword is used.

Time associations. In general, time associations are specified within the description of a field. There are duration keywords which can be used to indicate some time associations, like the frame_duration which indicates an instantaneous measurement in the example.

Drop outs. There are no conventions for handling drop outs.

Example Description

The metadata values at the beginning of the file provide the physical file attributes. The next set of keywords identify the source of the data and provide identification information for the data set and this particular data instance (all PDS data objects are grouped into data sets). All TIME fields in PDS labels use the ISO standard time format and are assumed to be in UTC. The ^TIME-SERIES label provides a pointer to the data file containing the time series. The OBJECT block describes the format of the fields in the time series, with individual column objects for each field. Note that the column objects provide explicit information about the storage format of the binary values in the data object (IEEE_REAL and MSB_INTEGER) which may be needed to interpret the data values in the future. Note that there is no vector data type for column values, so the position vector has to be handled as three separate fields. The count rate is handled as a column with 12 items and the items have a sampling parameter of 5 seconds each.

```
PDS_VERSION_ID                 = PDS3
RECORD_TYPE                    = FIXED_LENGTH
RECORD_BYTES                   = 76
FILE_RECORDS                   = 1440
HARDWARE_MODEL_ID              = 'SUN SPARC STATION'
OPERATING_SYSTEM_ID            = 'SUN OS 4.1.1'
^TIME_SERIES                   = 'V790101.DAT' /* FILE CONTAINING THE DATA*/
DATA_SET_ID                    = 'PVO-V-OMAG-4--SCCOORDS-24SEC-V1.0'
SPACECRAFT_NAME                = 'PIONEER VENUS ORBITER'
INSTRUMENT_NAME                = 'PARTICLE DETECTOR'
TARGET_NAME                    = VENUS
START_TIME                     = 1979-01-01T00:00:00.000Z
STOP_TIME                      = 1979-01-07T23:59:00.000Z
MISSION_PHASE_NAME             = 'VENUS ORBITAL OPERATIONS'
PRODUCT_ID                     = 'V790101.DAT'
PRODUCT_CREATION_TIME          = 1993-10-01
SPACECRAFT_CLOCK_START_COUNT   = '421532432'
SPACECRAFT_CLOCK_STOP_COUNT    = '421675654'

OBJECT                         = TIME_SERIES
TIME_SERIES_TYPE               = FEPC_TSO
KEY_FIELD                      = START_TIME
COLUMNS                        = 17
INTERCHANGE_FORMAT             = BINARY
ROW_BYTES                      = 76
ROWS                           = 10080

OBJECT                         = COLUMN
NAME                           = START_TIME
DESCRIPTION                    = "The start time for measurements in the
record.  Records normally occur every minute, however if there is missing
data there will be no records."
BYTES                          = 12
DATA_TYPE                      = ASCII
START_BYTE                     = 1
TIME_ORDER_TYPE                = ASCENDING
```

```
END_OBJECT                    = COLUMN

OBJECT                        = COLUMN
NAME                          = SC_LOCATION_X
DESCRIPTION                   = "X COMPONENT OF SPACECRAFT LOCATION."
BYTES                         = 4
DATA_TYPE                     = IEEE_REAL
START_BYTE                    = 13
END_OBJECT                    = COLUMN

OBJECT                        = COLUMN
NAME                          = SC_LOCATION_Y
DESCRIPTION                   = "Y COMPONENT OF SPACECRAFT LOCATION."
BYTES                         = 4
DATA_TYPE                     = IEEE_REAL
START_BYTE                    = 17
END_OBJECT                    = COLUMN

OBJECT                        = COLUMN
NAME                          = SC_LOCATION_Z
DESCRIPTION                   = "Z COMPONENT OF SPACECRAFT LOCATION."
BYTES                         = 4
DATA_TYPE                     = IEEE_REAL
START_BYTE                    = COLUMN

OBJECT                        = COLUMN
NAME                          = COUNT_RATE
DESCRIPTION                   = "HYPOTHETICALLY INSTANTANEOUSLY
DETERMINED COUNT RATE AT 12 ENERGY STEPS, DETERMINED AT THE RECORD'S TIME TAG
FOR THE FIRST COUNT_RATE WITH COUNT_RATES RATES DETERMINED AT 2RD-12TH ENERGY 2-
12 STEPS, EACH OFFSET BY 5 SEC FROM MEASUREMENT OF PRIOR STEP."
ITEMS                         = 12
ITEM_BYTES                    = 4
DATA_TYPE                     = MSB_INTEGER
START_BYTE                    = 25
SAMPLING_PARAMETER_NAME       = TIME
SAMPLING_PARAMETER_UNIT       = SECONDS
SAMPLING_PARAMETER_INTERVAL   = 5
FRAME_DURATION                = 0 <SEC>/* INSTANTANEOUS COUNT RATE */
/* OR SHOULD THE SAMPLING PARAMETER BE THE ENERGY STEPS??? */
END_OBJECT                    = COLUMN

OBJECT                        = COLUMN
NAME                          = DENSITY
DESCRIPTION                   = "DETERMINED FROM RATES 1-12 BY TAKING
MOMENTS OF DISTRIBUTION FUNCTION.  THE PROCESSING DURATION IS 55 SECONDS."
BYTES                         = 4
DATA_TYPE                     = IEEE_REAL
START_BYTE                    = 73
FRAME_DURATION                = 55 <SEC>
END_OBJECT                    = COLUMN
END_OBJECT                    = TIME_SERIES
END
```

Appendix C: Time-series API Detail

This section describes possible Application Programming Interfaces (APIs) that can be used to accommodate the Time Series Object (TSO) requirements defined in section 2. It has been estimated that for each of the formats discussed in section 3, the existing software support could likely be the basis for the types of services described below.

An Object Oriented (OO) approach is employed in designing the APIs with the following requirements and assumptions:

- Support simultaneous access to multiple TSOs.
- Data is organized and stored as described in Figure 1 (Time-Series Object Conceptual View) of section 2 , except the actual data values are viewed as trailing the time relationships for each record field.
- The structure of each TSO record is consistent

Sections C.1 and C.2 provide an introduction to the detailed API definitions specified in C.3 and C.4. A fully consistent set of API definitions has not been attempted, but there is enough to give a good idea of the approach.

Changes to be completed in future versions would include:

♦ In Section 2, we indicate that record sizes can vary. However some of the current API calls assume the record size is constant, i.e. each record (logically) contains an instance for every data field. If record size varied, then the data fields available in each record would vary. We may choose to provide an API either for vary size records or fixed sized records or perhaps even a API version for each case. The API version will affect what parameters are needed on inputs for many of the TSOField calls. We will likely need to update parameters for some of the calls whichever version of the API is provided.

♦ If data fields can vary within records or if the order of the data fields within a record are unknown, then additional data field identification and/or data type information will be needed in a number of the TSO field calls.

C.1 Application Programming Interfaces (APIs)

Each API returns a status code of Integer*4 to indicate whether the API is successfully completed. If the API is not successfully completed, it should return a negative number. Otherwise it should return the status code value of greater than or equal to 0 to indicate the successful completion of the API. Status codes and their explanation texts are not documented in this document since they can vary from implementation to implementation. The following describes the meaning for each of the data types used in the APIs.

Data Type	Description
String	A character string that has one or more characters. The size of a string is indicated by adding a colon after the word‘String’ followed by the string size. For example, String:20 represents a string that is 20 characters long.
Integer*4	4-byte signed integer
Real*4	4-byte floating point number
Void	Data type used to represent any of the data types described above. This data type is used to send or receive data to/from an API and is equivalent to ‘void’ in C, ‘equivalence’ in Fortran, and ‘Object” in Java.

There are two categories of APIs: low level APIs and high level APIs. Low level APIs provide a basic set of functions that allow users to create and manipulate TSOs, TSO records, and TSO record fields. High level APIs allow users to perform more sophisticated functions such as “get the records whose key time is between time A and time B”, “get the time range (begin and end time) for the given TSO”, “get the values for fields ’x’ and ‘y’ for the key time between time A and time B”, and etc. High level APIs make use of low level APIs as building blocks, often in combination with some programming.

Naming Conventions

- Calls are mixed case with the first letter of each word capitalized all other letters are lower case. (E.g., GetKeyTime, GetTSORec) (Note TSO is an abbreviation and is always capitalized.)
- Calling Parameters are mixed case, the first letter of each word except the first is capitalized, all other letters are lower case. (E.g., name, recNum, TSOId) (Note TSO is an abbreviation and is always capitalized.)
- The following abbreviations are used consistently throughout in names of Calls and Calling Parameters. These abbreviations are capitalized according to naming convention rules, except that TSO is always in all upper case.
    - Id - Identifier
    - Num - Number
    - Processing - Proc

    - Rec - Record
    - TSO - Time Series Object
- Calls named as CreateXxx will have a matching DeleteXxx call.
- Calls named as SetXxx will have a matching GetXxx call.

Low Level APIs

TSO Handling APIs

- CreateTSO (String:30 name, Integer*4 TSOId)
- DeleteTSO (Integer*4 TSOId)
- AskTSOId (String:30 name, Integer*4 TSOId)
- AskTSOName (Integer*4 TSOId, String:30 name)

TSO Record Handling APIs

- CreateTSORec (Integer*4 TSOId, Integer*4 recNum)
- DeleteTSORec (Integer*4 TSOId, Integer*4 recNum)
- AskTSORecSize (Integer*4 TSOId, Integer*4 recNum, Integer*4 recSize)
- SetKeyTime (Integer*4 TSOId, Integer*4 recNum, String:22 keyTime)
- GetKeyTime (Integer*4 TSOId, Integer*4 recNum, String:22 keyTime)
- SetTSORec (Integer*4 TSOId, Integer*4 recNum, String:variable TSORec)
- GetTSORec (Integer*4 TSOId, Integer*4 recNum, String:variable TSORec)

TSO Record Field Handling APIs

- CreateTSOField (Integer*4 TSOId, Integer*4 recNum, String:30 name,
  Integer*4 dataType, Integer*4 numDims, Integer*4 dimSizes[], Integer*4 fieldId)
- DeleteTSOField (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId)
- SetTSOField (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldID,
  Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)
- GetTSOField (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId,
  Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)
- SetTSOFieldData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldID,
  Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)
- GetTSOFieldData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId,
  Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)

- SetTimeOffset (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 timeOffset)
- GetTimeOffset (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 timeOffset)
- SetProcessingDuration Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 procDuration)
- GetProcessingDuration (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 procDuration)
- SetTimeRelationship (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, String:18 timeRelationship)
- GetTimeRelationship (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, String:18 timeRelationship)
- SetData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Void data)
- GetData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Void data)
- AskTSOFieldId (String:30 name, Integer*4 fieldId)
- AskTSOFieldName (Integer*4 fieldId, String:30 name)
- AskTSOFieldType (Integer*4 fieldId, Integer*4 dataType, Integer*4 numDims, Integer*4 dimSizes[])

High Level APIs

TSO Record Handling APIs

- ExtractTSORecs (Integer*4 TSOId, String:22 startTime, String:22 endTime,String:variable TSORecs[])
- AskTotalTSORecs (Integer*4 TSOId, Integer*4 numRecs)
- AskNumTSORecs (Integer*4 TSOId, String:22 startTime, String:22 endTime, Integer*4 numRecs)

TSO Record Field Handling APIs

- AskTSOFieldNames (Integer*4 TSOId, String:30 names[])
- AskTSOFieldIds (Integer*4 TSOId, Integer*4 fieldIds[])
- AskNumTSOFields (Integer*4 TSOId, Integer*4 numFields)
- AskNumDataValues (Integer*4 TSOId, String:22 startTime, String:22 endTime, String:30 fieldNames[], Integer*4 numDataValues)

- AskTSOFieldsData (Integer*4 TSOId, String:22 startTime,String:22 endTime, String:30 fieldNames[], Integer*4 numFields[], String:variable fieldData[])
- AskTSOFieldsData1 (Integer*4 TSOId, String:22 startTime,String:22 endTime, String:30 fieldNames[], Integer*4 numFields[], String:variable fieldData[])
- AskStartKeyTime (Integer*4 TSOId, String:22 startTime)
- AskEndKeyTime (Integer*4 TSOId, String:22 endTime)

C.2 Translation of Binary Data to String

A TSO record logically consists of a key time followed by a sequence of heterogeneous fields. If the user requests an entire TSO record, the requested record is returned in a contiguous block of octets. For convenience this API includes calls that return the data as its logical Data Type. However since the data type of one field can be different from other fields, the API also includes calls where the system translates individual field’s data into a common format (i.e. string) before returning the requested data. The following mapping guideline should be used for translating a binary data to a string for consistent implementation of the TSO:

Data Type	Translated String (left-justified)
1-byte, unsigned integer	3 bytes
1-byte, signed integer	4 bytes
2-byte, unsigned integer	5 bytes
2-byte, signed integer	6 bytes
4-byte, unsigned integer	10 bytes
4-byte, signed integer	11 bytes
4-byte, signed float (IEEE 754)	15 bytes
8-byte, signed float (IEEE 754)	25 bytes

As an example, if a field has two Integer*2 (signed) data values, say 1234 and 1244, then the returned value would be the 10 character string ‘1234 1244 '.

Throughout this API, when a key time is expressed as a string, it will be a 22 character string in the YYYYMMDDTHHMMSS.ssssss format where

YYYY = year,
MM   = month,
DD   = day,

T	= the character"T" (separates date and time portions
HH	= hour,
MM	= minute,
SS	= second
.	= decimal point separating the integer and fractional portion of the seconds (may be replaced by a space character if the entire fractional portion of a second is replaced)
ssssss	= fractional portion of second (any number of the trailing digits may be replaced an equivalent number of space characters)

When TSO Record Fields are input or output in String format, they will use the format shown below.

Field	Size (octets)	Comments
Time offset	15 [base is Real*4]	In seconds. Left-justified. Blank padded to correct length.
Processing duration	15 [base is Real*4]	In seconds. Left-justified. Blank padded to correct length.
Time relationship	18	Contains one of the following values: ‘start of duration ’ ‘middle of duration’ ‘end of duration ’ Note that the returned value is padded with blanks if the length of the string is shorter than 18.
Data	Variable	Size depends on how the field data was declared and stored.

When TSO Records are input or output in String format, they will use the format shown below.

Field	Size (octets)	Comments
Key time	22	The key time
Field *x* *(Repeat once for each field)*	Variable 48+data	Use field data format shown above for each field

C.3 Low Level APIs

Low level APIs provide a basic set of functions that allow users to create and manipulate TSOs, TSO records, and TSO record fields.

C.3.1 TSO Handling APIs

CreateTSO (String:30 name, Integer*4 TSOId)

It creates a TSO.

PARAMETERS:

name	String:30	In – the name of the TSO to be created
TSOId	Integer*4	Out – the identifier of the TSO just created

RETURNS:

status	Integer*4	Status code.

DeleteTSO (Integer*4 TSOId)

It deletes a TSO. All information associated with this TSO will be deleted (e.g. all records, including field data, key time, processing duration and time relationship, etc.).

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO to be deleted

RETURNS:

status	Integer*4	Status code.

<u>AskTSOId</u> (String:30 name, Integer*4 TSOId)

It returns the TSO ID for the TSO with the given name.

PARAMETERS:

name	String:30	In – the name of the TSO of interest.
TSOId	Integer*4	Out – the identifier of the TSO of interest

RETURNS:

status	Integer*4	Status code.

<u>AskTSOName</u> (Integer*4 TSOId, String:30 name)

It returns the name for the TSO with the given TSO identifier.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
name	String:30	Out – the name of the TSO of interest.

RETURNS:

status	Integer*4	Status code.

C.3.2 TSO Record Handling APIs

Note: The current API makes use of record numbers where we assume that records will be created in appropriate order and key times are monotonically added. Another option we could investigate in future issues would be to create records with given key and return simply record identifiers instead of record numbers. The API could then be responsible for maintaining the correct order. Note: We may want to provide an API call to adjust key time and offset of all fields within the record at the time.

<u>CreateTSORec</u> (Integer*4 TSOId, Integer*4 recNum)

It creates a TSO record.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
recNum	Integer*4	Out – the number of the record created. One (1) represents the first record

RETURNS:

status	Integer*4	Status code.

<u>DeleteTSORec</u> (Integer*4 TSOId, Integer*4 recNum)

It deletes a TSO record. All information associated with this TSO record will be deleted (e.g. all field data, key times, processing durations and time relationships, etc.).

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record

RETURNS:

status	Integer*4	Status code.

<u>AskTSORecSize</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 recSize)

Note: If we provide API calls for both varying and invariant records, we will need to update these parameters. If record sizes can vary, a record number is needed (and a call to get maximum record size is needed). If record sizes don't vary, then any record number should provide the same result, so record number is not needed.

It returns the TSO record size of the given TSO. The record size returned is the size in octets of the record returned by GetTSORec.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
recSize	Integer*4	Out – the record size for the given TSO record

RETURNS:

status	Integer*4	Status code.

<u>SetKeyTime</u> (Integer*4 TSOId, Integer*4 recNum, String:22 keyTime)

It sets the key time for the given record number.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
keyTime	String:22	In – the key time

RETURNS:

status	Integer*4	Status code.

<u>GetKeyTime</u> (Integer*4 TSOId, Integer*4 recNum, String:22 keyTime)

It returns the key time for the given record number.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
keyTime	String:22	Out – the key time for the given record number

RETURNS:

status	Integer*4	Status code.

<u>SetTSORec</u> (Integer*4 TSOId, Integer*4 recNum, String:variable TSORec)

It sets a TSO record for the given TSO ID and TSO record number. The record data is input in a contiguous block of octets, and it's the API's responsibility to map the input octets to the appropriate data types. The TSO record consists of a 'key time' followed by a sequence of field information (data, time offset, processing duration, time relationship).

Field information is repeated *n* times where n is the number of fields in a TSO record. For example, if there are two fields in a record, then the returned record would contain the following information:

Key time, Field 1 time offset, Field 1 processing duration, Field 1 time relationship, Field 1 data, Field 2 time offset, Field 2 processing duration, Field 2 time relationship, Field 2 data

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
TSORec	String:variable	In – TSO record for the given TSO ID and the TSO record number.

RETURNS:

status	Integer*4	Status code.

<u>GetTSORec</u> (Integer*4 TSOId, Integer*4 recNum, String:variable TSORec)

It returns a TSO record for the given TSO ID and TSO record number. The requested record is returned in a contiguous block of octets, and it's user's responsibility to map the returned octets into appropriate data types. The TSO record consists of a ‘key time’ followed by a sequence of field information (data, time offset, processing duration, time relationship.)

Field information is repeated *n* times where n is the number of fields in a TSO record. For example, if there are two fields in a record, then the returned record would contain the following information:

Key time, Field 1 time offset, Field 1 processing duration, Field 1 time relationship, Field 1 data, Field 2 time offset, Field 2 processing duration, Field 2 time relationship, Field 2 data

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
TSORec	String:variable	Out – TSO record for the given TSO ID and the TSO record number.

RETURNS:

status	Integer*4	Status code.

C.3.3 TSO Record Field Handling APIs

Note: This AIP makes it be possible to set time offset, processing duration, and time relationship separately. We may choose in future versions to require that all be reset if any one of them is reset since they are so closely linked.

CreateTSOField (Integer*4 TSOId, Integer*4 recNum, String:30 name, Integer*4 dataType, Integer*4 numDims, Integer*4 dimSizes[], Integer*4 fieldId)

It creates a TSO record field and returns a field ID.

PARAMETERS:

TSOId	Integer*4	In – identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
name	String:30	In - the name of the TSO record field to be created
dataType	Integer*4	In – data type of the field to be created Allowed values: TBD
numDims	Integer*4	In – number of array dimensions. Zero (0) represents that the data for this field is a scalar value. The value of 1 represents that the data is stored in a 1-dimenional array. The value of 2 represents that the data is stored in a 2-dimensional array, and so on.
dimSizes	Integer*4 []	In – array dimension sizes. The value of the first element of this array should be zero (0) if the data for this field is a scalar value. If the data is stored in a 10x3 2-dimensional array, then the values of the first and second element should be 10 and 3.
fieldId	Integer*4	Out – the identifier of the field just created.

RETURNS:

status	Integer*4	Status code.

<u>DeleteTSOField</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId)

It deletes a TSO record field. All information associated with this TSO field will be deleted (e.g. all time offsets, processing durations and time relationships, etc.).

PARAMETERS:

TSOId	Integer*4	In – identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record.
fieldId	Integer*4	In – the identifier of the field to be deleted.

RETURNS:

status	Integer*4	Status code.

<u>SetTSOField</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldID, Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)

It sets the field information (data, time offset, process duration, time relationship) in the given TSO record.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldID	Integer*4	In - the identifier of the field of interest
timeOffset	Real*4	In – time offset from the key time
procDuration	Real*4	In – processing duration
timeRelationship	String:18	In – time relationship of this field to the key time. It should be one of the following values: ‘start of duration’ ‘middle of duration’

		‘end of duration’
data	Void	In – the data values to be added to this field. The data can be a scalar, 1-dimensional array, or multi-dimensional array of any data type.

RETURNS:

status	Integer*4	Status code.

<u>GetTSOField</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 timeOffset, Real*4 procDuration, String:18 timeRelationship, Void data)

It returns the field information for the given field ID and record number. Field data, time offset, processing duration, and time relationship are returned.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
timeOffset	Real*4	Out – time offset from the key time.
procDuration	Real*4	Out - processing duration.
timeRelationship	String:18	Out – time relationship of this field to the key time that is one of the following values: ‘start of duration ’ ‘middle of duration’ ‘end of duration ’
data	Void	Out – data values for the given field id.

RETURNS:

status	Integer*4	Status code.

SetTSOFieldData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, String:variable data)

It sets the field information (time offset, processing duration, time relationship, and data) for the given field ID and record number. The field information is passed as a contiguous block of octets as described in the TSO field structure above.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
recNum	Integer*4	In – thenumber of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
data	String:variable	In – the field information

RETURNS:

status	Integer*4	Status code.

GetTSOFieldData (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, String:variable data)

It returns the field information (time offset, processing duration, time relationship, and data) for the given field ID and record number. The field information is passed as a contiguous block of octets as described in the TSO field structure above.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
data	String:variable	Out – the requested field information

RETURNS:

status	Integer*4	Status code.

<u>SetTimeOffset</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 timeOffset)

It sets the time offset for the given TSO, record number, and field.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
timeOffset	Real*4	In – time offset from the key time

RETURNS:

status	Integer*4	Status code.

<u>GetTimeOffset</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Real*4 timeOffset)

It returns the time offset for the given TSO, record number, and field.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
timeOffset	Real*4	Out – time offset from the key time

RETURNS:

status	Integer*4	Status code.

<u>SetProcessingDuration</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId,
Real*4 procDuration)

It sets the processing duration for the given TSO, record number, and field.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
procDuration	Real*4	In - processing duration

RETURNS:

status	Integer*4	Status code.

<u>GetProcessingDuration</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId,
Real*4 procDuration)

It returns the processing duration for the given TSO, record number, and field.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
procDuration	Real*4	Out - processing duration

RETURNS:

status	Integer*4	Status code.

SetTimeRelationship (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId,
String:18 timeRelationship)

It sets the time relationship for the given TSO, record number, and field.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
timeRelationship	String:18	In – time relationship of this field to the key time that has one of the following values: ‘start of duration ’ ‘middle of duration’ ‘end of duration ’ Note that the returned value should be padded with blanks if the length of the string is shorter than 18.

RETURNS:

status	Integer*4	Status code.

GetTimeRelationship (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId,
String:18 timeRelationship)

It returns the time relationship of the key time for the given TSO, record number, and field.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In – the identifier of the field of interest
timeRelationship	String:18	Out – time relationship of this field to the key time that has one of the following values: ‘start of duration ’ ‘middle of duration’ ‘end of duration ’ Note that the returned value is padded with blanks if the length of the string is shorter than 18.

RETURNS:

status	Integer*4	Status code.

<u>SetData</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Void data)

It sets the actual data field for the given TSO, record number, and field.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
data	Void	In – the data values to be added to this field. The data can be a scalar, 1-dimensional array, or multi-dimensional array of any data type.

RETURNS:

status	Integer*4	Status code.

<u>GetData</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldId, Void data)

It returns the actual data field for the given TSO, record number, and field.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
recNum	Integer*4	In – the number of the record of interest. One (1) represents the first record
fieldId	Integer*4	In - the identifier of the field of interest
data	Void	Out – the data values from this field. The data can be a scalar, 1-dimensional array, or multi-dimensional array of any data type.

RETURNS:

status	Integer*4	Status code.

Note: fieldId is considered to be universally unique in this version. Future version may reduce the scope of fieldId to be within a TSO. If that change is made a TSOId will need to be added to these calls.

AskTSOFieldId (String:30 name, Integer*4 fieldId)

It returns the field ID for the given TSO field name.

PARAMETERS:

name	String:30	In – the TSO field name to be searched
fieldId	Integer*4	Out – the identifier of the field of interest

RETURNS:

status	Integer*4	Status code.

[Note: This assumes fieldIds universally unique , i.e. TSOId and the record number are not needed.]

AskTSOFieldName (Integer*4 fieldId, String:30 name)

It returns the field name for the given TSO ID.

PARAMETERS:

fieldId	Integer*4	In – the identifier of the field of interest
name	String:30	Out – the TSO field name of interest

RETURNS:

status	Integer*4	Status code.

[Note: This assumes fieldIds universally unique , i.e. TSOId and the record number are not needed.]

<u>**AskTSOFieldType**</u> (Integer*4 fieldId, Integer*4 dataType, Integer*4 numDims,
Integer*4 dimSizes[])

It returns the data type and size for the given field ID.

PARAMETERS:

fieldID	Integer*4	In - the identifier of the field of interest
dataType	Integer*4	Out – data type
numDims	Integer*4	Out – number of array dimensions. Zero (0) represents that the data for this field is a scalar value. The values of 1 and 2 represent that the data is stored in a 1-dimenional array and 2-dimensional array, respectively.
dimSizes	Integer*4 []	Out – array dimension sizes. The value of the first element of this array should be zero (0) if the data for this field is a scalar value. If the data is stored in a 10x3 2-dimensional array, then the values of the first and second element should be 10 and 3.

RETURNS:

status	Integer*4	Status code.

C.4 High Level APIs

High level APIs allow users to perform sophisticated functions such as “get the records whose key time is between time A and time B”, “get the time range (begin and end time) for the given TSO”, “get the values for fields’x’ and ‘y’ for the key time between time A and time B”, and etc. High level APIs make use of low level APIs as building blocks, often in combination with some programming.

This section describes the APIs that address the commonly asked TSO questions by scientists. It contains the APIs identified to date, and by no means, it represents a complete set. As additional needs/functionalities are identified, this section will be modified to reflect the new requirements.

C.4.1 TSO Record Handling APIs

ExtractTSORecs (Integer*4 TSOId, String:22 startTime, String:22 endTime,
String:variable TSORecs[])

Note: The current version of the API could result in very large arrays of records being returned. A future version may refine the API to simply create a new TSO with a different TSOId to jointly refer to the required subset of TSO records. Then the other TSO record calls could be used to process that subset.

It returns the TSO records whose key time falls between the user-specified start key time and end key time in an array of TSO records (see the description of the GetTSORec API in section 4.2 for a detailed description of the TSO record). Note that it's user's responsibility to map the returned TSO record(s) into appropriate data types.

The number of records returned from the AskNumTSORecs call should be used in allocating the needed space for the TSOrecs[] argument.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
startTime	String:22	In – user-specified start key time
endTime	String:22	In – user-specified end key time
TSORecs	String:variable []	Out – an array of TSO records that satisfied the user-specified search criteria

RETURNS:

status	Integer*4	Status code.

<u>AskTotalTSORecs</u> (Integer*4 TSOId, Integer*4 numRecs)

It returns the total number of TSO records stored in the given TSO.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
numRecs	Integer*4	Out - the total number of records in the given TSO

RETURNS:

status	Integer*4	Status code.

<u>AskNumTSORecs</u> (Integer*4 TSOId, String:22 startTime, String:22 endTime, Integer*4 numRecs)

It returns the number of TSO records whose key time falls between the user-specified start key time and end key time.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
startTime	String:22	In – user-specified start key time
endTime	String:22	In – user-specified end key time
numRecs	Integer*4	Out - the number of records that satisfied the user-specified search criteria

RETURNS:

status	Integer*4	Status code.

Note: If future versions of the API move toward using record identifiers rather than record numbers, calls to move to next and previous record (based on key times) would be useful.

AskTSORecNext (Integer*4 TSOId, Integer*4 recIdIn, Integer*4 recIdOut)

AskTSORecPrev (Integer*4 TSOId, Integer*4 recIdIn, Integer*4 recIdOut)

C.4.2 TSO Record Field Handling APIs

AskTSOFieldNames (Integer*4 TSOId, String:30 names[])

It returns the TSO field names defined for the given TSO.

The number of TSO fields returned from the AskNumTSOFields call should be used in allocating the needed space for the names[] argument.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
names	String:30 []	Out – an array that contains the TSO field names that are defined for the given TSO.

RETURNS:

status	Integer*4	Status code.

AskTSOFieldIds (Integer*4 TSOId, Integer*4 fieldIds[])

It returns the TSO field IDs defined for the given TSO.

The number of TSO fields returned from the AskNumTSOFields call should be used in allocating the needed space for the fieldIds[] argument.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest
fieldIds	Integer*4 []	Out – an array that contains the identifiers of the fields of interest.

RETURNS:

status	Integer*4	Status code.

<u>AskNumTSOFields</u> (Integer*4 TSOId, Integer*4 numFields)

It returns the total number of TSO fields defined for the given TSO.

PARAMETERS:

TSOId	Integer*4	In – the identifier of the TSO of interest.
numFields	Integer*4	Out – the number of fields defined for the given TSO

RETURNS:

status	Integer*4	Status code.

<u>AskNumDataValues</u> (Integer*4 TSOId, String:22 startTime, String:22 endTime,
String:30 fieldNames[], Integer*4 numDataValues)

It determines and returns the total number of user-specified fields data values that fall between the user-specified start key time and end key time.

This number should be used in allocating the needed space for the fieldData[] argument of the AskTSOFieldsData and AskTSOFieldsData1 APIs.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
startTime	String:22	In – user-specified start key time
endTime	String:22	In – user-specified end key time
fieldNames	String:30 []	In – an array that contains the field names to be searched. Each element of the array represents a field name to be searched.
numDataValues	Integer*4	Out – the total number of user-specified field data values that fall between the user-specified start key time and end key time and return that number

RETURNS:

status	Integer*4	Status code.

<u>AskTSOFieldsData</u> (Integer*4 TSOId, String:22 startTime,String:22 endTime,
String:30 fieldNames[], Integer*4 numFields[], String:variable fieldData[])

It returns the user-specified field data whose record key time falls between the user-specified start key time and end key time.

The number of data values returned from the AskNumDataValues call should be used in allocating the needed space for the fieldNames[], numFields[], and fieldData[] arguments.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
startTime	Sting:22	In – user-specified start key time
endTime	String:22	In – user-specified end key time
fieldNames	String:30 []	In – an array that contains the field names to be searched. Each element of the array represents a field name to be searched.
numFields	Integer*4 []	Out – the number of data values extracted for each field defined in fieldNames. The value of the 1st element represents the number of data values extracted for the first field defined in

		fieldNames, 2nd element represents the number of data values extracted for the second field defined in fieldNames, and so on.
fieldData	String:variable []	Out – extracted field data values. The values are returned in the following format:

Array position	Represents
1 – numFields[1]	fieldNames[1]
numFields[1]+1 through numFields[1]+numFields[2]	fieldNames[2]
.	
.	
numFields[n-1]+1 through numFields[n-1]+numFields[n]	fieldNames[n]

Note:

- The size of the array element is determined by the largest data type of user-specified fields.

RETURNS:

status	Integer*4	Status code.

<u>AskTSOFieldsData1</u> (Integer*4 TSOId, String:22 startTime,String:22 endTime,
String:30 fieldNames[], Integer*4 numFields[], String:variable fieldData[])

It returns the user-specified field data whose record key time plus field 'time offset' falls between the user-specified start key time and end key time.

The number of data values returned from the AskNumDataValues call should be used in allocating the needed space for the FieldNames[], numFields[], and fieldData[] arguments.

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
startTime	String:22	In – user-specified start key time
endTime	String:22	In – user-specified end key time
fieldNames	String:30 []	In – an array that contains the field names to be searched. Each element of the array represents a field name to be searched.
numFields	Integer*4 []	Out – the number of data values extracted for each field defined in fieldNames. The value of the 1st element represents the number of data values extracted for the first field defined in fieldNames, 2nd element represents the number of data values extracted for the second field defined in fieldNames, and so on.
fieldData	String:variable []	Out – extracted field data values. The values are returned in the following format:

Array position	Represents
1 – numFields[1]	fieldNames[1]
numFields[1]+1 through numFields[1]+numFields[2]	fieldNames[2]
.	
.	
numFields[n-1]+1 through numFields[n-1]+numFields[n]	fieldNames[n]

Note:

· The size of the array element is determined by the largest data type of user-specified fields.

RETURNS:

status	Integer*4	Status code.

<u>AskStartKeyTime</u> (Integer*4 TSOId, String:22 startTime)

It returns the start key time for the given TSO. This time is the key time retrieved from the first record.
(Convenience routine for
  GetKeyTime(TSOId, 1, startTime)

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
startTime	String:22	Out – the key time of the first record.

RETURNS:

status	Integer*4	Status code.

<u>AskEndKeyTime</u> (Integer*4 TSOId, String:22 endTime)

It returns the end key time for the given TSO. This time is the key time retrieved from the last record.
(Convenience routine for
  GetTotalTSORecs(TSOId, numRecsTemp)
  GetKeyTime(TSOId, numRecsTemp, endTime)

PARAMETERS:

TSOId	Integer*4	In - the identifier of the TSO of interest
endTime	String:22	Out – the key time of the last record.

RETURNS:

status	Integer*4	Status code.

Note: If future versions of the API move toward using varying record sizes or varying field orders within a TSO record, then calls to move to next and previous fields would be useful.

<u>AskTSOFieldNext</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldIdIn,
Integer*4 fieldIdOut)

<u>AskTSOFieldPrev</u> (Integer*4 TSOId, Integer*4 recNum, Integer*4 fieldIdIn,
Integer*4 fieldIdOut)

======